Programmation sûre en précision finie : Contrôler les erreurs et les fuites d'informations

Ivan Gazeau

Safe Programming in Finite Precision: Controlling Errors and Information Leaks

Thèse de Doctorat

Spécialité informatique

présentée et soutenue publiquement

le 14 octobre 2013 par

IVAN GAZEAU

Devant le jury composé de :

Directeurs de thèse :	Dale Miller	(INRIA, École Polytechnique)
	Catuscia Palamidessi	(INRIA, École Polytechnique)
Rapporteurs :	Michele Boreale	(University of Florence, Italy)
	Herbert Wiklicky	(Imperial College, UK)
Examinateurs :	Eric Goubault	(CEA)
	Olivier Boissou	(CEA)

Travaux effectués au Laboratoire d'Informatique de l'École Polytechnique

ACKNOWLEDGMENTS

I would like to thanks Dale for his support all along my PhD. He has been a great advisor who provides me insights about where to find deep material. He was always available to hear about my progress and new investigations and support me to burry on my ideas. I also would like to thanks Catuscia for her investment in my PhD. She provide me very good insights as weel as she has been very pedagogic with me, especially about the way to introduce and to present my work. The topics adressed during my PhD were quite broad, so I really appreaciate the complementarity of the expertise of Dale and Catuscia that allow me to get a large perspective.

I am honoured that Herbert Wicklicky and Michele Boreales have accepted to review my thesis and I thank them for their careful reading.

I am delighted that Olivier Buissou and Eric Goubault accepted to be part of my jury. This thesis was partly founded by the ANR project Confidence Proof and Provability, where I had the pleasure to meet Olivier Buissou who was a charismatic organisor and Eric Goubault who spends a lot of time to introduce me to the present challenges of finite-precision arithmetics .

I also thanks Jean Goubault-Larrecq that was co-organizer of the ANR project for his advices and expertize on my research project. I worked for a while with Filippo Bonchi, even if at the end, we did not concretize our work, I have been very pleased to exchange ideas with him.

Because scientific work is not possible without a warm and convivial environment, I would like to thanks all former and later members of the Parsifal and the Comete team. I have a special thought for my parents and family that encourage me all along my PhD, I also warmly thanks my roomates Cyrille, Gilles and Olivier for the nice three years of life together and I lovely thanks Isabelle for everything.

Finally, I would like to thank you, dear reader, for your attention on my thesis work.

CONTENTS

RÉSUMÉ EN FRANÇAIS

Contrairement à l'homme qui manipule des valeurs symboliques lors de ses calculs ($\sqrt{2}$, π, etc.), un ordinateur calcule à partir d'approximation des nombres. En effet, même s'il est possible, de nos jours, d'implémenter le calcul symbolique, celui-ci reste coûteux en ressources et est beaucoup plus lent qu'un calcul direct à base d'approximations. Bien entendu, le fait d'arrondir les nombres provoque des erreurs de calculs, et même si tout est fait au niveau de l'architecture des processeurs pour rendre ces erreurs aussi négligeables que possible, celles-ci peuvent, dans certaines circonstances, perturber fortement le résultat.

Ces erreurs d'arrondis dont certaines sont restées tristement célèbres (explosion de la fusée Ariane 5, explosion d'un missile sur une mauvaise cible) ont toujours été et font toujours l'objet d'études approfondies. Comme les preuves manuelles de correction ont toujours le risque d'être fausses et requerraient trop de travail pour des logiciels industriels comportant des milliers de lignes de code, on a développé des programmes pour automatiser l'analyse des programmes.

L'analyse de programmes est appelée statique car le programme n'est pas exécuté, contrairement aux analyses à base de tests. Cela lui permet de fournir une garantie totale de la robustesse du programmes aux erreurs (contrairement aux tests qui, généralement, ne sont pas exhaustifs et sont donc faillibles). Dans cette thèse cependant, nous ne cherchons pas à développer une méthode d'analyse automatique ou semi-automatique mais nous cherchons à relier des preuves de corrections d'algorithmes à des analyses automatiques de code. En effet, dès que le principe d'un programme est non trivial, il repose souvent sur un théorème qui prouve que l'algorithme va fournir le bon résultat. Ce théorème peut potentiellement utiliser des arguments mathématiques complexes et sera donc la plupart du temps hors d'atteinte d'un analyseur automatique. Mais malheureusement, les théorèmes sont souvent prouvés uniquement pour un calcul exact et ne permettent pas de prédire quantitativement l'influence des erreurs d'arrondis lors du calcul. Le but de cette thèse est de proposer pour deux classes d'algorithmes des théorèmes étendus qui, en se basant sur la preuve exacte et sur une analyse simple du code, permettent de quantifier l'erreur finale résultant des arrondis.

Les deux problèmes que nous avons abordés sont les suivants. D'abord nous nous sommes intéressé à la question de la confidentialité différentielle (*differential privacy*). Il s'agit d'un

mécanisme permettant à un individu de participer à une étude statistique sans que l'on puisse retrouver ses données personnelles. L'intérêt de ce problème est que le programme ne calcule pas la valeur d'une fonction mais produit une valeur (pseudo)-aléatoire. Or, comme un analyseur automatique ne fournit qu'une borne sur la déviation maximale d'un calcul, il faut réussir à réinterpréter cette déviation en terme de variation de la loi de probabilité.

Dans notre second problème, on s'intéresse à des programmes où les branchements conditionnels modifient radicalement la façon dont on atteint le résultat. Dans de tels programmes, une infime variation dans les valeurs peut conduire le programme à emprunter deux branches dont les effets sont radicalement différents. Bien sûr, de tels programmes fonctionnent car, au final, il y a une sorte de convergence qui s'opère. Cependant, un analyseur fait progresser son analyse ligne après ligne par compositions successives et ne peut pas appréhender cette convergence globale. De fait, quand il se retrouve face à deux branches distinctes qui mènent à des résultats différents, il risque de simplement conclure que le programme est instable. Pour palier à ce problème nous proposons un théorème qui utilise la propriété de convergence prouvée pour des calculs exacts ainsi qu'une analyse du programme qui ne se soucie pas des branchements pour conclure un résultat de robustesse pour le programme en calcul arrondis.

0.1 Robustesse

Représentation finie

Avant de prouver la fiabilité d'un programme, il est important de comprendre en quoi consiste exactement le calcul en représentation finie. Principalement, nous nous intéressons à deux types de représentations finies : les nombres à virgules flottantes et les nombres à virgules fixes.

Les nombres à virgules fixes sont utilisés soit sur des processeurs rudimentaires soit pour des calculs où l'amplitude des nombres est connue à l'avance (la monnaie par exemple) soit dans quelques cas où ils sont plus performants que les nombres à virgule flottante. Leur représentation consistent principalement en un entier divisé par une constante prédéfinie. Lors d'un calcul en virgule fixe, les additions et soustractions se font sans erreur car le résultat est aussi représentable. Le principal problème de cette représentation, c'est qu'elle ne permet pas de représenter de très grand nombres donc une multiplication peut facilement déclencher un dépassement de capacité.

Les nombres à virgules flottantes sont plus élaborées car ils possèdent un exposant variable et non constant (d'où le nom de flottant). Ces nombres étant les plus utilisés, leur comportement est strictement encadrée par le standard IEEE 754. Les opérations primitives (addition, soustraction, multiplication, division, logarithme, exponentielle, passage à l'exposant) sont garantis pour avoir

un résultat avec une précision totale: le résultat (arrondi) est tel qu'il n'existe aucun autre nombre représentable entre lui et le résultat exact. En d'autres termes, l'erreur vient juste de l'arrondi, il n'y a pas d'erreur de calcul. Cependant, cette propriété n'est vraie que pour une opération. Dès qu'un programme va enchaîner un nouveau calcul sur une opération déjà arrondie, l'arrondi final ne sera pas forcément optimal.

Analyse statique

Le but de cette thèse est de proposer des outils qui viennent en aval d'une première analyse statique du code, aussi, nous décrivons brièvement leur principe de fonctionnement ainsi que les enjeux de ces analyses.

Il y a deux familles de méthodes généralement utilisées. D'une part il y a les preuves logiques basées sur les triplets de Hoare. Cette méthode reprend les techniques utilisées pour prouver les formules logiques. Un triplet de Hoare consiste en une précondition, un élément de syntaxe du code à étudier et une postcondition. L'autre famille d'analyseur est basée sur l'interprétation abstraite. Dans ce cas, on définit à l'avance un domaine abstrait qui, en regroupant plusieurs états de la mémoire en un seul, permet de prédire des propriétés du résultat pour n'importe quel entrée.

Indépendamment de la méthode employée, il faut définir ce qu'on entend par robustesse. La première définition est une relation entre l'entrée et la sortie. On souhaite que de petites perturbations de l'entrée n'aient pas d'impact majeur sur la sortie du programme. Dans le cas contraire, une erreur de mesure ou une erreur provenant d'un calcul précédent générerait une erreur arbitraire indépendamment de la précision du programme lui-même.

La deuxième propriété à considérer est celle de l'erreur interne causée par le programme. Tandis que la question de l'entrée / sortie est inhérente à la fonction qu'on calcule, l'erreur interne se définit entre la sémantique exacte du programme et sa sémantique effective avec erreur.

Enfin, nous soulevons la question de la gestion de plusieurs variables. Comme nous souhaitons une modélisation la plus souple possible, nous introduisons la notion de mesure qui permet de gérer de façon simple et uniforme un ensemble arbitraire de variables.

Notre approche

A partir de ces analyses sur ce qu'est un nombre en représentation finie et ce que fait une analyse statique de programme, nous justifions deux définitions de la robustesse que nous utiliserons par la suite. En effet, la plupart des propriétés que fournit un analyseur automatique permettent de déduire les propriétés que nous proposons. De plus, celles-ci sont simples à manipuler mathématiquement et permettent ainsi de se combiner facilement aux preuves du programme dans la

sémantique exacte. La première des deux définitions, la propriété $P(k,\varepsilon)$, concerne uniquement la question des entrées/sorties. Tandis que la seconde, la propriété de (k,ε)-promixité, concerne l'écart entre la sémantique exacte et la sémantique effective.

0.2 Confidentialité différentielle

L'analyse statique permet, dans son usage le plus fréquent, de mesurer l'erreur induite par le programme et permet, par exemple, d'informer l'utilisateur du programme du nombre de décimales qui ne sont pas affectées par ces erreurs. Dans le chapitre 3, nous nous intéressons aux problèmes de l'erreur dans un contexte différent. Ici, il s'agit de comprendre l'impact des erreurs de calculs pour un programme utilisé dans la protection des données. Précisément, on peut vouloir fournir un résultat à partir de données qui doivent rester confidentielles. Dans ce cas, le problème n'est pas vraiment que le résultat soit imprécis mais, avant tout, qu'une corrélation entre la nature de l'erreur et les entrées peut permettre à celui qui reçoit le résultat d'obtenir des informations sur les entrées confidentielles. Dans le cas qui nous intéresse, la confidentialité différentielle, nous montrons que son implémentation stricte, telle que spécifiée théoriquement pour un calcul exact, va provoquer des failles importantes. Il est donc nécessaire d'adapter l'implémentation pour empêcher ces failles. Ensuite, il s'agit de montrer que le nouveau protocole est sûr malgré les erreurs de calcul. Plus précisément, il s'agit de mesurer la perte provoquée par les erreurs d'arrondis et de montrer que celle-ci est acceptable. Après avoir montré brièvement le type de faille qui peuvent apparaître, l'essentiel de ce chapitre se concentre sur une modélisation du problème qui soit la plus indépendante possible de la représentation finie utilisée ainsi que du programme lui-même (on suppose juste qu'il répond aux spécifications sans présupposer du code lui-même). A partir de cette modélisation, nous établissons, via un théorème, la perte en confidentialité induite par les erreurs. Enfin, nous montrons deux applications possibles de ce théorème suivant le type de données à protéger.

Description de la confidentialité différentielle

La confidentialité différentielle est une approche ayant pour but de garantir la confidentialité des données des participants à une étude statistique. En effet, une base de donnée à usage statistique doit révéler des informations générales sur les participants telles que des moyennes ou l'existence de certaines corrélations. Cependant, la plupart des participants à ses bases de données n'acceptent de livrer leur données que si on leur promet que leurs informations personnelles ne seront pas divulguées. Ainsi, il faut pouvoir, par exemple, être en mesure de donner le pourcentage de la population qui fume sans révéler qui, précisément, fume.

Pour satisfaire ces deux contraintes, la protection de la vie privée et la publication d'informations générales, de nombreuses méthodes ont été proposées. La première a été de se contenter d'anonymiser les identifiants personnels des participants puis de transmettre les données ainsi anonymisées aux analystes qui sont alors libres de faire les requêtes qu'ils souhaitent. Une telle méthode ne permet cependant pas de garantir l'anonymat: rien ne garantit en effet qu'un analyste ne possède pas déjà une partie des informations de la base de donnée. Par exemple, dans une base médicale, même si on supprime les nom, prénom et numéro de sécurité sociale des participants, on conservera leur âge selon toute vraisemblance. Supposons maintenant que la doyenne des français participe à l'étude. Étant probablement la seule de son âge, il est facile pour un analyste de trouver à quelle entrée elle correspond et de connaître ainsi tout son dossier médical. On peut chercher, par exemple, la proportion de diabétiques parmi les personnes de plus de 112 ans.

Ce type d'attaque peut sembler simple à prévenir, aussi d'autres méthodes d'anonymisation ont vu le jour pour parer au problème. Cependant, par des attaques plus complexes ces autres méthodes se sont elles aussi avérées vulnérables. La confidentialité différentielle repose sur un protocole plus strict que ses prédécesseurs : aucune partie de la base de données n'est jamais fournit à l'analyste. Au lieu de cela, l'analyste doit, pour chaque requête, contacter la base de données qui est détenue par un agent de confiance. Celui-ci répond aux requêtes de façon probabiliste en ajoutant de l'aléatoire dans sa réponse. La propriété de confidentialité différentielle stipule que la probabilité d'obtenir une réponse donnée à une requête est sensiblement la même qu'une personne donnée participe ou non à la base de donnée. Autrement dit, on ne risque rien à participer à la base de donnée puisque si on ne participait pas, les analystes obtiendraient des réponses qui seraient indiscernables à celles obtenues en cas de participation.

Pour obtenir cette propriété, il faut que le facteur aléatoire soit proportionnel à la sensibilité de la requête à la présence d'un individu supplémentaire quelconque. De la sorte, si une requête cible trop précisément un individu, le bruit ajouté sera suffisamment important pour masquer l'information.

L'implémentation théorique et ses faiblesses en pratique

Pour obtenir cette propriété, la méthode théorique consiste à calculer le résultat réel de la requête puis d'ajouter une valeur aléatoire répartie selon une distribution adéquate (par exemple la distribution de Laplace) dont l'amplitude dépendra du degré de confidentialité recherché et de la sensibilité de la requête.

Cette méthode fonctionne en théorie mais elle s'appuie sur le fait qu'il existe des probabilités de distribution continue (deux valeurs proches ont des probabilités quasiment identiques

d'apparaître). Elle repose également sur le fait qu'il est possible qu'une variable aléatoire puisse produire des valeurs arbitrairement grandes avec une probabilité arbitrairement faible. Or ces deux propriétés ne sont plus conservées lorsqu'on génère une valeur pseudo-aléatoire en précision finie. En effet, à cause des arrondis (et, a fortiori, en cas d'erreur de calcul) certaines valeurs précisent peuvent ne pas être représentée du tout tandis qu'une valeur proche peut apparaître avec une probabilité plus élevée qu'elle ne devrait (voir figure 3.1). De cette façon, il n'y a plus de continuité dans la distribution de la probabilité. Nous montrons dans l'exemple 3.1.1 comment ce défaut permet d'accéder à des informations confidentielles.

Modélisation et analyse de la situation en précision finie

Dans la partie technique du chapitre 3, nous montrons d'abord comment sont générées les variables pseudo-aléatoires. Nous montrons que celles-ci peuvent être assimilées à des variables aléatoires parfaites qui ont subit une perturbation (non contrôlée celles-ci) d'amplitude bornée.

Ensuite nous proposons une amélioration du protocole qui va palier aux défauts décrits précédemment: pour éviter les irrégularités locales de la distribution, on impose que le résultat retourné soit arrondi plus que la précision de la machine. Pour palier au fait qu'une représentation finie ne peut produire des nombres arbitrairement grands avec une probabilité arbitrairement faible, on impose de retourner une erreur si le résultat produit dépassait une certaine borne. Enfin, on traduit la déviation en terme de distance entre distributions de probabilité. Après avoir rajouté une dernière contrainte quant à la forme de la distribution utilisée, on prouve que la perte de confidentialité est bornée par une constante calculable.

On montre ensuite comment le théorème s'applique dans le cas standard le plus simple. On montre que dans certains cas, malgré les contraintes supplémentaires, la perte peut demeurer importante. On propose alors une autre façon de générer les nombres pseudo aléatoire pour ce cas qui réduit drastiquement cette perte. Enfin on montre que notre analyse permet aussi de traiter des données plus complexes comme des bases de données ayant des données à valeur dans le plan euclidien.

0.3 Analyse globale d'un programme

Dans le chapitre 4, nous nous intéressons à l'analyse du programme elle-même, à savoir borner l'erreur provoquée par la représentation finie d'un programme donné. Comme nous l'avons précédemment évoqué, il existe déjà plusieurs façon d'analyser automatiquement un code donné. Cependant ces méthodes fonctionnent de façon progressives. En quelque sorte, elles extraient une information sur cette borne pour les n premières lignes du code à partir de laquelle elles

fournissent une information sur les $n+1$ lignes de code. Et ainsi de suite jusqu'à atteindre la fin du programme.

Ici on s'intéresse à des programmes qui ne peuvent pas être analysés de cette façon. En effet, certains programmes produisent une erreur importante en cours d'exécution qui ne se résorbe qu'à la fin de l'exécution. C'est le cas par exemple des programmes qui effectuent une dichotomie pour trouver une valeur : au moment où on teste si la valeur cherchée est supérieure ou inférieure, une erreur de calcul peut mener dans l'autre branche du test. Dans ce cas, le programmes va poursuivre avec des valeurs très différentes de celles qu'il aurait dû prendre mais la valeur finale va tout de même s'approcher de la valeur exacte. Pour illustrer notre analyse sur ce type de phénomène, on a pris comme exemple l'algorithme CORDIC qui calcule la fonction sinus. Un autre cas est celui où l'ordre dans lequel on effectue une tâche permet un certain parallélisme: à cause des erreurs de calcul, on peut choisir de commencer par une tâche plutôt que par une autre (et donc avoir un état intermédiaire très différent de l'état théorique) mais, à la fin, une fois les deux tâches effectuées le résultat est quasiment identique. Pour illustrer cet autre type de phénomène, nous avons choisi d'étudier l'algorithme de Dijkstra qui calcule le plus court chemin dans un graphe.

La méthode d'analyse que nous proposons utilise la preuve mathématique qu'un algorithme est correct pour borner l'erreur que peut générer son implémentation en représentation finie. Par exemple, dans le cas de CORDIC, on va utiliser le fait qu'avec des calculs exacts l'algorithme produit une fonction continue de dérivée bornée (pour le calcul de la fonction sinus). Cependant cette propriété ne peut pas être exploitée directement car la preuve part du principe, entre autres, que des rotations d'un point autour d'un même axe vont conserver la distance de ce point à l'axe. Cela ne sera plus le cas dans l'implémentation: les erreurs de calculs pourront aussi translater ce point et l'éloigner ou le rapprocher de l'axe.

En plus de cette preuve en sémantique exacte, nous décomposons le programme selon un motif prédéfini qui va nous permettre, d'une part, d'analyser chacune des parties isolément via les méthodes classiques existantes et, d'autre part, d'analyser le programme en terme de système de réécriture. En effet, les systèmes de réécritures possèdent des outils efficaces pour traiter de la confluence (le fait qu'un comportement puisse diverger mais ne puisse atteindre qu'un seul état final).

Cette analyse par décomposition est donc une méthode globale qui permet de calculer l'erreur liée aux arrondis et nous montrons qu'elle fonctionne aussi bien avec l'algorithme CORDIC qu'avec l'algorithme de Dijkstra.

Conclusion

Dans cette thèse, nous nous intéressons aux rapports qu'entretiennent les résultats théoriques exacts avec leur implémentation en précision finie. Nous avons traité de ce problème dans le cadre de la confidentialité différentielle ainsi que dans le cadre d'une méthode d'analyse globale se basant sur la preuve en calcul exact. Dans les deux cas, nous montrons qu'il est possible d'étendre les résultats exacts aux résultats approchés mais que cela nécessite une adaptation préalable. Ainsi, en confidentialité différentielle, l'implémentation directe conduit à des failles majeures. Pour garantir la validité du résultat théorique en précision finie l'algorithme doit être renforcé. En ce qui concerne l'analyse globale, celle-ci nécessite, en plus de la stabilité mathématique de la fonction programmée, une analyse plus fine des propriétés de confluence du système de réécriture sous-jacent.

⋆

CHAPTER 1

INTRODUCTION

Contents

Computers have been created to replace slide rules and manual computations. Unlike human beings, they do not make careless mistakes, they compute must faster and they never get tired. Now that producing processors has become very cheap and their size has became very small, computers are used everywhere for any purpose.

However, computers are not intelligent robot that can program themselves. Their automated computations are conceived by engineers. These developers rely most of the time on some mathematical results that state that some equations or algorithms should return the intended results.

Unfortunately, mathematical results are obtained with symbolic computations: for instance, we can directly states that $(\cos x)^2 + (\sin x)^2 = 1$ without even knowing the value of x. Even if some programs allows symbolic computations, these programs are resource consuming and they cannot be used in most cases. So, instead of computing with mathematical values, computers make approximations of numbers and provide approximate results. Fortunately, these approximate results are very close to the exact ones so that in most cases, not having the exact result is acceptable. Sometimes, however, the result can be too different to be useful. It may occur, for instance, when a program makes long computations: errors stack and can become non negligible with respect to the true answer. But, sometimes, errors can be critical even for a short program in case a little shift leads to a completely different option. This can happen because of

some conditional instructions that can lead the program into different branches depending on a comparison of an approximate value with some threshold. This may also happen because the result is analyzed by a nasty agent who succeed in retrieving initial confidential inputs from the nature of the error of the output.

In this setting, how can we be confident in a program given that it does not behave exactly as it should? There are several approaches to answer this question. At one extreme, the developer implements his program from some mathematical results then adapt the theorem such that it remains true even if there is some deviation. Such kind of technique is heavy since it requires hand proof, and since such proofs can be quite long there is a strong risk of errors. However, this method is the most flexible: the manual proof allows to use complex arguments that no automated algorithm can produce. At the other extreme, one can try to develop a static analyzer that reads the code and that quantifies the error without any human intervention. Such an approach is mandatory when the code to check is a large software package with millions of code lines. This method, however, has some drawbacks. First, a generic algorithm for analysis has only generic rules to apply to the code: it cannot use subtle arguments. Secondly, an automatic proof is almost never readable. Indeed, when the problem becomes too complex, the analyser splits the input domain which can lead to combinatory explosion. Finally, even if it succeeds in finding the proof, the proof is not well structured as a human proof and does not help to understand any general principle behind the proof.

In this thesis, we develop an approach that falls in between the above approaches. We model errors at a high level such that it is possible to get simple mathematical statements about them. Then we provide theorems that start from the mathematical proof of the algorithm on which we add some additional conditions about errors. These new theorems provide a little weaker results than the initial one but they grant properties about the actual computation, not on the exact result.

Our results are quite general, in the sense that they apply to a large class of algorithms, and are not tied to a particular implementation.

More precisely, we have studied two classes of programs and we have proved they are safe. The first class consists of programs that implement differential privacy, a new technique to hide personal information when providing the results of a survey. In this case, our high level modeling allows us to provide, from information about the maximal deviation, information about the shift on a probabilistic distribution. The other class consists of programs which have some kind of erratic behavior during the execution but that, at the very end, renders a good approximation of the exact result.

Our results are based on assumptions that are not guaranteed by all algorithms. However, they are general enough to capture many interesting programs, which makes worthwhile to develop proof methods.

1.1 Contributions and plan of the thesis

Our first contribution consists in providing two definitions to deal with deviation errors due to finite representation. The first one, the $P(k,\varepsilon)$ property, allows to define functions which are robust in a finite precision setting and which are less constraining than usual mathematical definitions. The second one is the (k,ε)-*closeness* property between two functions. This property allows us to grant that the function in the finite-precision semantics is never far away from the exact function.

Our first main result is about differential privacy. When differential privacy protocols are designed, the question of the finite representation was, until recently, not considered as a sensitive source of leakage. However, we show that the straightforward transcription of the algorithms in a finite representation architecture leads to a protocol that critically leaks information. This problem was also studied, independently, in [Mir12]. That paper provides guarantees for a given implementation of the most used function. Here, we provide a general method to analyze any noise function that aims at implementing differential privacy. By modeling errors as a shift in the probability distribution, we are able to prove that the addition of some safeguards prevents massive leakage. In addition, we also measure the additional leakage induced by rounding errors depending on the amplitude of the error and on the function which is implemented. As a side result, we also propose an improved algorithm in the standard case of the Laplacian noise in one dimension.

The second main result is a theorem to prove that a class of algorithms based on some "global behavior" can be safely implemented in finite representations. These algorithms cannot be studied by standard methods because they rely on subtle arguments. Indeed, small deviations can totally change the control flow and leads to completely different values in internal states. We mainly study two characteristic examples: the CORDIC algorithm to compute trigonometric function and the Dijktra's algorithm to find the shortest distance between two nodes of a graph. As a first try, we expose in section 4.3 a direct method to analyze the robustness of the code without considering the behavior of the exact algorithm. The theorem proves the $P(k,\varepsilon)$ property for the program. This theorem relies on an underlying program transformation proof.

Since we found the application to the example not straightforward, we develop a second method in section 4.4 that considers the relation between the exact and the finite precision semantics. The method consists in interpreting the control flow as a non deterministic process and then we use techniques based on rewrite abstract systems to prove the global confluence of the program. The new theorem states the closeness property between the exact and the finite-precision semantics. The theorem is based on the hypothesis that the exact semantics as already been stated robust ($P(k,\varepsilon)$) and requires some additional hypothesis. This theorem is highly non

trivial since the regularity is not implied by the stopping condition like in a classical convergent algorithm and since the proof in the exact semantics relies on invariants that are broken in the finite representation.

The thesis consists of three chapters. In chapter 2, we start by a technical introduction presenting notations and usual definitions about computations and errors as well as techniques for analysis of programs. Then we introduce our approach of robustness: our definition and the properties they enjoy. Chapter 3 is about differential privacy. We start by a technical introduction specific to this domain. Then we present our method to analyze the leakage due to finite representation of numbers. In chapter 4, we present our work on programs that are locally discontinuous while they are robust as a global function.

1.2 Publications

This thesis is based on two published papers and on some unpublished work.

- The first article [GMP12a], by Dale Miller, Catuscia Palamidessi and myself, was presented in QAPL 2012.
- The second article [GMP12b], by the same authors, is in course of submission to a journal. This article is an extension of the former one because it considers the same problem but the methodology and techniques used are new material.
- The third article [GMP13] by the same authors was presented in QAPL 2013.

Chapter 2 starts with a technical introduction then Section 2.3 is based on the articles [GMP12a, GMP12b]. Chapter 3 is mostly based on the article [GMP13]. However, the improvement proposed in section 3.6.3, is an unpublished material. Chapter 4 is based on the articles [GMP12a, GMP12b]. The subsection 4.4.6 is a little extension based on an idea of a reviewer.

$\star$

CHAPTER 2

ROBUSTNESS

Contents

In this chapter, we are interested in the definition and the study of robustness in a generic way. Informally, robustness is a property that indicates that the program "behaves well" even when it is exposed to perturbations. Here, the considered perturbations are the ones due to finite precision. Finite precision means that real numbers are approximated so that they can be stored in a finite memory space.

Such approximation is a central problem in computer science. It has been well studied and it is still an active research area. In this thesis, we are looking for new theoretical techniques to find solutions for specials cases where standard methods cannot apply.

In this thesis, we do not aim at analyzing code from scratch. Such an analysis actually requires a strong control of the properties of the program and is dependent on the finite representation used. Here, we start from the analysis made by an existing analyzer and we use the result either to prove another property (the differential privacy property in chapter 3) or to analyze a program more complex than the analyzed one (in chapter 4 we prove the correctness of the whole program while we start from the analysis of some parts). So, we need a definition of robustness general enough so that most analyzers can be able to prove it and that is not too technical to be able to use it in theorems.

In the literature, there are several definitions of robustness that have been considered and many of them are mathematical properties about classical functions. These definitions coined for pure mathematical statements are not suitable for the problem we examine here. The goal of this chapter is to explain why, and to propose other definitions.

In the first section, we start by describing the behavior of a computer and the basic definitions to formalize such a behavior. In the next section, we explain what is program analysis and what is our policy about it. Finally, the last section contains our contribution: it consists of a definition about the regularity of function and of a definition that expresses how close to the exact semantics is the finite-precision semantics. We also provide various properties of these definitions.

2.1 Hardware and semantics preliminaries

At the hardware level, an instruction is a function from bits to bits where bits are binary values. To represent more complex structures than sequences of bits, types are defined: they allow to provide an interpretation of the meaning of each bits as well as constraints to manipulate them.

Processors can do a fixed number of operations on a succession of bits. An n bit processor is a processor that operates with n bits simultaneously. Most processors, presently, are either 32 bits or 64 bits. When a computation requires more than these n bits, the operation is split into several successive operations.

In any case, we do not have infinite sequences of bits to represent reals. So, apart from exact computations based on symbolic representation of numbers, real numbers have to be rounded. There exists mainly two kinds of representation for real numbers that mostly depends on the processor. The floating-point representation, that requires a processor with a specific module (the floating point unit) is the most used. The other one is the fixed-point representation. It is used on low-cost processors which do not have floating-point units and also, sometimes, when floating point representation is not suitable (computations on currencies, for instance).

2.1.1 The fixed-point representation

In this simple representation, each value is stored in n bits. Let $i_0, \ldots, i_{n-1}$ denote these n bits. In this representation the number i represented is $\pm i_{n-2} \ldots i_d . i_{d-1} \ldots i_0$ where d is the number of bits used for the fractional part of the number. Hence the set of representable numbers is

$$\mathbb{D} = \{z \cdot 2^{-d} | z \in [-2^{n-1}, 2^{n-1}]\}.$$

With this representation, numerical operations work in the same way as for integers. The advantage of this representation is that processing is easy. On the other hand, it can be used only when the number to manipulate are all in the same range $\mathbb{D}$, so that there is no need to use a dynamic exponent.

$$
\begin{array}{cccccccccc}
 & & & 1 & 1 \cdot 1 & 0 & 1 & & & \\
* & & & 1 & 0 \cdot 0 & 1 & 0 & & & \\
\hline
 & & & & \cdot 1 & 1 & 1 & 0 & 1 \\
+ & 1 & 1 & 1 & 0 \cdot 1 & & & & & \\
\hline
 & 1 & 1 & 1 & 1 \cdot 0 & 1 & 1 & 0 & 1
\end{array}
$$

Figure 2.1: A multiplication in fixed-point representation, as taught in school.

The fixed-point arithmetic The main advantage of fixed-point representation is that addition and subtraction are done without any error. The only case in which an error can happen is when the sum of two numbers exceeds the maximal capacity. Technically this is not a rounding error but rather an overflow error.

The multiplication of two fixed-point numbers generates both rounding and overflow error as illustrated in figure 2.1. The rounding is an inherent problem of finite precision representation. But, here it can easily lead to an underflow exception: the result is rounded to 0. Rounding to 0 a non zero number is problematic since it cannot be used to divide a number while it is possible with the exact number. On the other hand, the overflow (i.e. the loss of the most significant bits) is the most serious problem of this representation. Indeed, the returned number is not related anymore to the exact number, so that most of the time it leads to a fatal error. These two problems, underflow and overflow, are permanent problems to the developer since the amplitude of representable numbers is just 2^n which is very limiting.

2.1.2 The floating-point representation

Representation of reals in an efficient way is a central problem in computer science. Reals are manipulated so often that the processor contains a part dedicated to manipulate them. This part is called the floating point unit (FPU). Since errors on representation of reals are a big issue, this unit has been fully optimized and therefore it is more complex to explain than fixed point representation.

As we explained, there are two main problems with fixed-point representation. On one hand, to define their type, we need an additional parameter: the number of digits that stands for the fractional part. On the other hand, avoiding overflow and underflow errors requires a special attention. To avoid these two issues, the floating-point representation allocates some bits to

define the exponent.

There are several kinds of floating-point representations that are used depending on their size. Here, we detail only the most used one called "doubles" which stands for double precision floating-point number.

Doubles use 64 bits: one bit s for the sign, another one e_0 for the sign of the exponent, 10 bits $e_1,\ldots,e_{11}$ for the absolute value of the exponent and 52 bits $m_1,\ldots,m_{52}$ for the mantissa. The number d represented in this way is:

$$d = (-1)^s \left(1 + \sum_{i=1}^{52} m_i 2^{-i}\right) e^{(-1)^{e_0} \sum_{j=1}^{10} e_j 2^j}$$

In addition, there are some additional combinations of bits that represent errors like "infinite result" when a division by zero occurs.

Full precision for atomic operators in floating-point representation There are two kinds of rounding errors. The first ones is intrinsic to the finite precision representation, i.e., any time the result of a computation would be a non representable number then it has to be rounded to a representable number. For instance, $\sqrt{2}$ cannot be represented by a finite number. The other kind of error is due to a wrong result for the computation, i.e., the returned result is not the closest representable number of the mathematical result.

Definition 2.1.1 (full precision)**.** *An implementation of a function is "full precision" if there is no representable number in between the provided result and the exact one.*

The main goal that motivated the creation of floating-point numbers was to provide full precision for all mathematical operators. Another objective was to be able to specify which one of the two admissible results (the greater or the lower one) has to be returned such that the programmer can fully determine which result will be returned. These two goals have been achieved years ago: the IEEE standard 754 [IEE08] for floating-point arithmetic certifies that the returned result for any one step operation is one of the closest numbers that the floating point can represent. This standard also proposes several policies for rounding: to return the greater or the lower number, to return the closest to zero number or to return the number closest to the true result.

Addition and subtraction While fixed point representation adds no rounding error for addition, floating point numbers rounds result each time the ratio a/b is outside of $[-0.5,-2]$. Indeed, let e_a and e_b the two exponents of a and b that are added. If $e_a = e_b$ (and a and b have the same

sign) then the final exponent will be $e_a + 1$, so the last bit is lost. If $e_a > e_b$ then either $e_{a+b} = e_a$ or $e_{a+b} = e_a + 1$. In that case, the last $e_a - e_b$ bits of b are not used to compute $a + b$.

Multiplication and division Multiplicative operators have a much better implementation with floating-point than with fixed-point representations since the exponent will change in order to always keep the most significant bits. So, in this representation, there is no useless zeros or strong bit truncation. The result is not always the exact one however. A rounding happens at least each time we divide by a number such that the rational result has a quotient which is not a power of two. In addition, the multiplication of the two mantissas is a multiplication of two integers: the returned result is twice as long as the initial numbers. So, since the size of the mantissa is also 53, half of the least significant bits are rounded.

Exponentiation Exponentiation is also considered as a primitive by the IEEE standard. However, the actual computation is done through three steps according to the following formula:

$$x^y = 2^{\log_2(x)y}$$

To be able to provide full precision for this composed operations, processors use a larger internal floating-point representation with 80 bits such that rounding errors remain small and do not affect the final result.

2.1.3 Semantics of programs. Definitions and notations

As we have seen, the actual problem of rounding numbers does not occur when there is just one operation that is performed: the floating-point semantics grants that the error is negligible proportionally to the value. As we will see, the situation is different for a whole program. But in order to speak about programs we need to introduce first some definitions and notations.

Programming language Programming languages allow programmers to produce machine code without worrying about the actual machine instructions of a specific processor. The more elaborate is the language, the more high-level is the code written by the programmer. In our study, we choose to describe programs through a pseudo language, so that we can highlight the arithmetic operations.

We represent functions with the following syntax:

```
f(x,y){
  ...
  return z;
```

```
}
```

Here `f` is he name of the function and `x,y` are the parameter of the function. The instruction `return z;` indicates that the function returns `z` (which has to be defined in the body of the program).

For instance, the following code is an implementation of the square function.

```
square(x){
  return x * x;
}
```

The following code, is also an implementation of the square function.

```
square(x){
  y=(x+1);
  return x * y - x;
}
```

In general, given a mathematical function, there is not a unique implementation of it. While two implementations are equivalent in theory, their rounding errors have no reason to be the same.

To make the link between a program code and a mathematical function, we use an inductive definition called denotational semantics. We use the notation $[\![\texttt{instructions;}]\!]$ to represent the semantics of the code instruction. The base case are the variables and the constants. The semantics of a variable `x` is its value (we will describe later which one). The semantics of a composite expression, like `x + y`, is obtained from the semantics of the components. For instance, $[\![\texttt{x + y}]\!] = [\![\texttt{x}]\!] + [\![\texttt{y}]\!]$.

This definition is non ambiguous, while we consider there is no rounding. But, for our purpose we need to differentiate between the intended computation i.e. where operations are made on reals and the actual computation where operations are the ones described by the finite-precision arithmetic. To distinguish between them, we note $[\![\texttt{x + y}]\!]'$ the result of the computation in the finite precision system.

When we have already set that $[\![\texttt{instructions;}]\!] = f$ where f is a function, we will use the more readable notation f' instead of $[\![\texttt{instructions;}]\!]'$ such that $[\![\texttt{instructions;}]\!]' = f'$.

2.1.4 Problems induced by the rounding error

We have briefly presented finite-precision implementations and the semantics of programs. Now, we present some of the challenges raised by floating-point representation. Indeed, while a single

operation grants the most accurate result, a succession of several operations may lead to critical deviations.

Difficulty to compute the exact error Finite representations do not enjoy traditional properties [Gol91]. In particular, the associativity of the addition is broken in floating-point representation. Indeed, when computing $(a+b)+c$, first $a+b$ is rounded then c is added, while for computing $a+(b+c)$, $(b+c)$ is rounded first. So, if $b=-c$ and $a \ll b$, $a+b$ is rounded to b such that the final result is 0 , while $b+c=0$ so the result of `a+(b+c)` is a.

This kind of errors is unpredictable for the developer since compiler optimizations may change the order the operations are executed. Thus, if the programmer writes in a high-level programing language, it might be impossible to predict the errors that the compiled code will make. That is one of the reasons why we do not try to compute the error but just to compute a bound on it.

Absolute or relative error? When dealing with rounding errors, we would like to ensure a property like "the error never exceeds p percents of the exact number". Such a property, however, is not compositional. Indeed, as we have seen in the previous example, if an addition is made between two numbers a and b such that the ratio b/a is big, then adding $-b$ increases the relative error significantly. Since such a phenomenon is not easily traceable (except in case of a simple addition of positive numbers), we prefer in our study to concentrate on absolute errors (i.e. we do not provide a percentage but the deviation itself).

Absolute errors are harder to compute when an algorithm is mostly based on multiplications. Indeed, if we know some bounds ε on the absolute error on a and b then the error after the multiplication is $a\varepsilon+b\varepsilon+\varepsilon^2$, which is not bounded as long as a and b are not bounded. However, in the algorithms we study this is not problematic because we never do iterative multiplications and because there always exists a maximal bound on all inputs values. In addition, we present an example to show that in some cases relative errors are also hard to compute and can lead to weaker results than absolute errors.

Example 2.1.1 (Robustness breaks with a simple iterative loop)**.** *In this example, we stress that non robust behaviors (i.e., behaviors vulnerable to rounding errors) can emerge even from simple and regular programs. Consider the following code.*

```
geo(x0,x1,n){
  u=x0;
  v=x1;
  w=0;
```

```
    for(i=0 to n){
        w= a*v + b*u;
        u= v;
        v= w;
    }
  return w;
}
```

This code, in the exact semantics, computes the value u_{n+1} of the sequence defined by: $u_0 = x0$, $u_1 = x1$ and

$$u_{n+1} = au_n + bu_{n-1}$$

A mathematical theorem states that

$$u_n = Ar_1^n + Br_2^n$$

where r_1 and r_2 are the distinct roots of the quadratic

$$X^2 - aX - b$$

and A and B are fixed by the initial condition.

Now, assume that a and b are representable numbers such that $r_1 < 1$ and $r_2 > 1$. Then, assume we run this program with parameter $x0 = 1$ and $x1 = r_1$. In that case, we have $A = 1$ and $B = 0$. So, in the exact semantics, a big value of n provide a result close to 0. In the finite precision semantics however, if r_1 is not representable then the input will be an approximation r_1'. Since the initial conditions are changed the B value is not zero anymore but ε. If n is big, the term in Ar_1^n tend to zero while the term in εr_2^n due to errors tend to infinity.

Finally, it is not possible to efficiently bound the relative error of this program due to just one critical case, while the absolute error is pretty regular for all values.

2.2 Quantitative analysis

A quantitative analysis means that we measure the error made by the finite representations. There are several choices to do this measure. Before presenting our definitions, we explain the reason of our choice.

Our goal in chapter 3 is to provide a method to compute the probabilistic information leakage of a program once a quantitative analysis of the error has already been performed. In chapter 4, we provide a method such that once some parts of the code have been analyzed then it is

possible to analyze the whole program from the point of view of its robustness to errors. We are interested in some formulation of robustness which can be performed by traditional tools and which can be easily manipulated from a mathematical point of view.

2.2.1 Static analysis

The term "static" means that the analysis is done without executing the program but just by processing the code of the program. Static analysis provides over approximations of the studied property. In our case, this means that a bound on the error is not necessary the optimal one. A static analysis is better than a battery of test cases since it provides a result valid for the whole domain while dynamic testing may fail to test some cases.

There are two main methods to do a static analysis of a program, whatever the analysis is about. We briefly present them.

Logic based proof This technique has been introduce by Hoare [Hoa69] and has now several variants. The principle of this analysis consists on considering any instruction as a transformation between preconditions and post conditions. The main feature of Hoare logic are the Hoare triple that describe the execution of one instruction. A Hoare triple is written as:

$$\{P\}I\{Q\}$$

where P is a formula that is the precondition, I is the instruction and Q is the post condition.

From these triples it is possible to derive other triples with inference rules. For instance, the following rule allows to derive a triple for a sequence of instructions.

$$\frac{\{P\}I_1\{Q\} \qquad \{Q\}I_2\{R\}}{\{P\}I_1;I_2\{R\}}$$

Abstract interpretation Abstract interpretation [CC77] aims at providing over approximations of all possible behaviors of a program. In abstract interpretation, the semantics of a program, i.e., $[\![.]\!]'$, is called the concrete semantics. The domain whose elements constitute the over representations of values and states of programs is called "abstract domain". For instance, an abstraction of the exact value of a variable can be an interval in which this variable belongs. Once an abstract domain is defined, the abstract interpretation computes the abstraction that results from the previous abstraction by applying the instruction. For instance, if our abstract domain consists of intervals, from an interval $[a,b]$ the instruction `x=2*x;` leads to the interval $[2a,2b]$.

2.2.2 Input-output relation

There are several kinds of properties that may be interesting for studying the errors due to finite precision. The first question when we study error is the sensitivity of the output for small variations of the input. Indeed, even if a program computes the result in full precision, the input value, in general, comes from either a physical value or another computation and is not the exact value. If the program has an erratic behavior then even a small error on the input leads to a big error to the output.

A weak property that can be defined about the relation between input and output is continuity:

Definition 2.2.1 (continuity)**.** *A function $f : \mathbb{R} \to \mathbb{R}$ is continuous, if*

$$\forall \varepsilon > 0 \, \forall x, x' \in \mathbb{R} \, \exists \delta \;\; |x - x'| \leq \delta \implies |f(x) - f(x')| \leq \varepsilon$$

The continuity property ensures that the correct output can be approximated when we can approximate the input closely enough. This notion of robustness, however, is too weak in many settings, because a small variation in the input can cause an unbounded change in the output.

On the other hand, a function which is not continuous has little chance to be robust. Since various regularity properties imply continuity, it is sometime useful to prove a function is not continuous in order to avoid to waist time trying to prove these properties.

A stronger property is the k-Lipschitz property.

Definition 2.2.2 (k-Lipschitz)**.** *A function $f : \mathbb{R}^m \to \mathbb{R}^n$ is k-Lipschitz, according to the distances d_m and d_n, respectively, if*

$$\forall x, x' \in \mathbb{R}^m \;\; d_n(f(x), f(x')) \leq k \cdot d_m(x, x')$$

The k-Lipschitz property amends this problem because it fixes a bound on the variation of the output linearly to the variation of the input.

The k-Lipschitz property has been used by Chaudhuri et al [CGL10, CGLN11] to define robustness. To prove a program to be k-Lipschitz, they first prove the function is continuous then they prove the function is piecewise k-Lipschitz while they compute k .

However, the k-Lipschitz property does not deal with algorithms that have a desired precision e as a parameter and are considered correct as long as the result differs by at most e from the results of the mathematical function they are meant to implement. A program of this kind may be discontinuous (and therefore not k-Lipschitz) even if it is considered to be a correct implementation of a k-Lipschitz function.

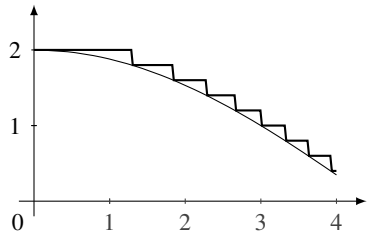

Figure 2.2: The function g^{-1} and its approximation

Example 2.2.1. *The phenomenon is illustrated by the following program f which is meant to compute the inverse of a strictly increasing function $g : \mathbb{R}^+ \to \mathbb{R}^+$ whose inverse is k-Lipschitz for some k.*

```
f(i){ y=0;
      while(g(y) < i){
            y = y+e; }
      return y; }
```

The program f approximates g^{-1} with precision e in the sense that

$$\forall x \in \mathbb{R}^+ \; f(x) - e \leq g^{-1}(x) \leq f(x)$$

Given the above inequality, we would like to consider the program f as robust, even though the function it computes is discontinuous (and hence not k-Lipschitz, for any k). We illustrate in figure 2.2 how the function cos *is approximated by such an algorithm.*

This example motivates our definition of input output relation that we will define later.

2.2.3 Dealing with several variables

The previous definitions are valid only for functions with one argument and one output. However, to be general, we should give definitions that are applicable to programs with several arguments.

In general, a program may have an arbitrary number of arguments. For instance, a program sorting a list deals with all the numbers the list contains. Another example is a program receiving a stream of physical data every second. These two examples are not analyzed in the same way however. Indeed, a program that receives a stream and agglomerates the results (like a program that sums all entries) has a fixed memory size. In that case, we can consider that the program has only one variable that receives all the values, as illustrated by the following code.

```
sum(){
  y = 0 ;
  while(input not empty)
    y = y + new_input();
  return y;
```

In that case, we can consider the program only has one internal variable `y` while the input variables are the outputs of the `new_input()` instruction. With this interpretation, inputs errors are seen as internal errors of the instruction `new_input()` instead of input errors of the program.

The case where the program can use an arbitrary large memory is more difficult. However, the memory capacity of any device is limited and allocation of too much memory can crash the program. So, to prove the robustness of the program, it is necessary to prove that this allocation will be bounded, and therefore we can only consider the cases up to the maximal size. On the other hand, dynamic allocation slows down the execution of the program. Since robustness analysis is mostly made for critical system like embedded system that have to be fast and simple, these kind of programs with dynamic allocation are not really studied and we do not study them either.

Therefore, we restrict our study to programs that deal with a fixed number of variables. To be able to do an accurate analysis of the propagation of error, it is interesting to track each variable individually.

For instance, an approach used by Majumdar et al in [MS09, MSW10] consists on formulating robustness with the following definition. A function f is (δ, ε)-robust on the i-th input, if a variation of at most δ on the i-th variable while all other are identical, makes a shift of at most ε.

Another approach has been considered to deal with error annihilation when computations add then remove the same quantity. For instance, the expression $3y - y$ can only double the initial error from y, while $3y - x$ can have an error which is the sum of the deviations of $3y$ and x.

Studying how robustness depends on each variable is very useful to get better approximations of the error. However in our case, we use results coming from automated analysis to obtain either a more complex property than just error deviation (the differential privacy in chapter 3) or a property about a more complex program (in chapter 4). To be able to use any kind of former analysis which can be more or less precise and to provide general results, we need a general enough definition. That is why we prefer to use the notion of distance that aggregates errors from several variable into one quantity. Formally, a distance is defined as follows:

Definition 2.2.3 (Metric space)**.** *A metric space is an ordered pair* $(\mathbb{M}, d)$ *where* $\mathbb{M}$ *is a set and* d *is a distance on M, i.e., a function*

$$d\colon \mathbb{M}\times\mathbb{M}\to\mathbb{R}$$

such that for any $x,y,z\in M$, the following holds:

1. $d(x,y)\geq 0$ *(non-negative),*
2. $d(x,y)=0 \iff x=y$ *(identity of indiscernibles),*
3. $d(x,y)=d(y,x)$ *(symmetry) and*
4. $d(x,z)\leq d(x,y)+d(y,z)$ *(triangle inequality) .*

Since we are interested in programs computing with real valued inputs and outputs, we mostly use metric spaces based on $\mathbb{R}^m$, the cross product of $\mathbb{R}$ m times where $m\in\mathbb{N}$.

The set $\mathbb{R}^m$ is a normed vectorial space [Rud86]. This means we can add two vectors, we can multiply any vector a vector by a scalar (an element of $\mathbb{R}$) and there is a norm on it. In the case of $\mathbb{R}^m$, if $x=(x_1,\ldots,x_m)$ and $y=(y_1,\ldots,y_m)$ are in $\mathbb{R}^m$, then $x+y$ is defined components by components :$x+y=(x_1+y_1,\ldots,x_m+y_m)$. The scalar multiplication is done the same way: $\lambda x=(\lambda x_1,\ldots,\lambda x_m)$. We also use the notation $x-y$ that stands for $x+(-1)\cdot y$.

Finally, norms are defined as follows.

Definition 2.2.4 (L_p norm)**.** *For $n\in\mathbb{N}$ and $x=(x_1,\ldots,x_n)\in\mathbb{R}^m$, the L_p norm of x, which we will denote by $\|x\|_p$, is defined as*

$$\|x\|_p=\sqrt[p]{\sum_{i=1}^{n}|x_i|^p}$$

From these norms, it is possible to define all natural distances (the one that only rely on the normed vectorial space structure).

Definition 2.2.5 (distance)**.** *The distance function corresponding to the L_p norm is*

$$d_p(x,y)=\|x-y\|_p.$$

We extend this norm and distance to $p=\infty$ in the usual way:

$$\|x\|_\infty=\max_{i\in\{1,\ldots,n\}}|x_i|$$

and

$$d_\infty(x,y)=\|x-y\|_\infty.$$

When clear from the context, we will omit the parameter p and write simply $\|x\|$ and $d(x,y)$ for $\|x\|_p$ and $d_p(x,y)$, respectively.

2.2.4 Internal rounding error

The relationship between inputs and outputs does not give any information on how big the computational error can be inside the program itself. If we prove that the exact semantics of a program is k-Lipschitz, it means that if the input contains an error e then the result amplifies this error by at most k. However, an arbitrary large error (i.e., not function of e) can still appear as a consequence of rounding errors associated to the computation.

On the other hand, even if we prove that the finite-precision semantics of the program is k-Lipschitz, this does not mean that the computation is close to the exact function. We can imagine for instance a program that, due to errors, returns always 0 while the function is supposed to return a less trivial result.Such a program is 0-Lipschitz but is not robust.

In general, to know a robustness condition between inputs and outputs is necessary to have a robust program, but it is not sufficient. So, to analyze internal errors, we need to develop other specifications than just a property about the function. Mainly, we need to consider a relation between both the exact semantics and the finite precision semantics of the program. Both abstract interpretation [GP11] and Hoare based logic [BF07] study internal errors through pairs consisting of the value in the exact semantics and the finite precision semantics.

In abstract interpretation, the concrete semantics is a triple (r,f,e) where r is the real value (in the exact semantics), f the actual floating point value and e is the error ($e = f - r$). The abstract domain consists of the values of the following form:

$$f = r + \sum_i a_i \varepsilon_i$$

where the ε_i represent any value in the interval $[-1,1]$ and the a_i are real-valued scale factors.

In the Hoare logic, the preconditions and the post conditions are properties that state the maximal distance between the two semantics.

Since the former approach also implies a maximal distance (the sum of the a_i) and that distances are nice to be manipulated mathematically, we will consider the following definition.

Definition 2.2.6 (L_∞ norm (on functions))**.** *The L_∞ norm of a function $f\colon \mathbb{R}^m \to \mathbb{R}^n$, according to a given norm L_p on $\mathbb{R}^n$, is defined as:*

$$\|f\|_\infty = \sup_{x \in \mathbb{R}^m} \|f(x)\|_p.$$

This definition makes an implicit reference on the norm $\|\cdot\|_p$ used to measure single vector. In our study, we will mostly use the $\|\cdot\|_1$ norm. From the last definition, we can also define the infinite distance between two functions.

Definition 2.2.7. *Let f and f' two functions $\mathbb{R}^m \to \mathbb{R}^n$,*

$$d_\infty(f,f') = \sup_{x\in\mathbb{R}^m} \|f(x) - f(x)\|_p.$$

2.3 Our approach to robustness

In this section, we provide the definitions we will use in the next chapters as well as the properties that these definitions enjoy.

2.3.1 Input-output relation

First, we consider only the relationship between inputs and outputs of the function. As we have explained in section 2.2.2, some work in the literature consider the k-Lipschitz property. However, example 2.2.1 illustrates an example where a function which is not k-Lipschitz has to be considered robust because the discontinuity are less than some small value. In [BF07], they amend this problem by considering a third semantics in addition to the exact and the finite precision one: the so-called *intended* semantics. Here, we prefer not to introduce another semantics but to consider that the implemented function is close up to ε to a regular function.

Definitions

The definition we propose is the following.

Definition 2.3.1 (The $P(k,\varepsilon)$ property)**.** *Let $(\mathbb{R}^m, d_m)$ and $(\mathbb{R}^n, d_n)$ two metric spaces. Let $f : \mathbb{R}^m \to \mathbb{R}^n$, $k, \varepsilon \in \mathbb{R}^+ \cup \{0\}$, we say that f is $P(k,\varepsilon)$ if*

$$\forall x, x' \in \mathbb{R}^m, d_n(f(x), f(x')) \leq k \cdot d_m(x,x') + \varepsilon$$

This property is a generalization of the k-Lipschitz property, which can be expressed as $P_{k,0}$. In general, for $\varepsilon > 0$, $P(k,\varepsilon)$ is less strict than k-Lipschitz and avoids the problem illustrated in example 2.2.1. The function in that example,in fact, is $P_{k,e}$.

In some cases, a large deviation on the input may cause the deviation on the output to go out of control, independently from "how well" the program behaves on small deviations. In general, we hope to maintain the deviation small, hence it is useful to relax the $P(k,\varepsilon)$ definition to consider only inputs that are quite close (up to some δ). So we provide the following $P(k,\varepsilon,\delta)$ definition:

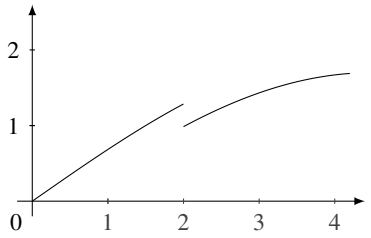

Figure 2.3: A $P(k,\varepsilon)$ function

Definition 2.3.2 (The property $P(k,\varepsilon,\delta)$)**.** *Let $(\mathbb{R}^m,d_m)$ and $(\mathbb{R}^n,d_n)$ be metric spaces , $f:\mathbb{R}^m\to\mathbb{R}^n$ be a function, $k,\varepsilon\in\mathbb{R}^+$, and let $\delta\in\mathbb{R}^+\cup\{+\infty\}$. We say that f is $P(k,\varepsilon)$ if:*

$$\forall x,x'\in\mathbb{R}^m,\quad d_m(x,x')\leq\delta\implies d_n(f(x),f(x'))\leq kd_m(x,x')+\varepsilon$$

This new definition is an extension of the former one since $P(k,\varepsilon)=P(k,\varepsilon,\infty)$.

Properties of our notions of robustness

We, now, provide some useful properties about the above definitions. First, we relate the $P(k,\varepsilon)$ property to the k-Lipschitz property.

Proposition 2.3.1. *If f is k-Lipschitz and $\|f'-f\|_\infty\leq\varepsilon$, then f' is $P(k,2\varepsilon)$.*

Proof. Consider the triangular inequality

$$d_n(f'(x),f'(y))\leq d_n(f'(x),f(x))+d_n(f(x),f(y))+d_n(f(y),f'(y))$$

Then since f is k-Lipschitz:

$$d_n(f'(x),f'(y))\leq d_n(f'(x),f(x))+kd_m(x,y)+d_n(f(y),f'(y))$$

Finally, since $\|f'-f\|_\infty\leq\varepsilon$:

$$d_n(f'(x),f'(y))\leq\varepsilon+kd_m(x,y)+\varepsilon$$

which is the definition of $P(k,2\varepsilon)$. □

Regular functions in the intended semantics are k-Lipschitz, we introduced the P(k,ε) property to weaken the k-Lipschitz property in order to also accept functions that are close to their intended semantics. This property states that functions close up to ε to a k-Lipschitz intended function are actually P$(k,2\varepsilon)$.

The next property allows to compose two functions that are $P(k,\varepsilon)$. This composition is less optimized than the mechanisms described to do static analysis. Indeed, our metric does not keep any trace of the correlations between variables or about the particular influence of each variable. However, it is still useful to achieve composition at a mathematical level between parts of programs.

Proposition 2.3.2 (Compositionality)**.** *If $f : \mathbb{R}^n \to \mathbb{R}^q$ and $g : \mathbb{R}^m \to \mathbb{R}^n$ are $P(k_1,\varepsilon_1)$ and $P(k_2,\varepsilon_2)$ respectively, then $f \circ g$ is $P(k_1k_2, \varepsilon_1 + k_1\varepsilon_2)$*

Proof. Since f is $P(k_1,\varepsilon_1)$ we have:

$$d_q(f(g(x)), f(g(y))) \leq k_1 d_n(g(x), g(y)) + \varepsilon_1$$

Since g is $P(k_1,\varepsilon_1)$ we have:

$$d_g(f(g(x)), f(g(y))) \leq k_1(k_2 d_m(x,y) + \varepsilon_2) + \varepsilon_1$$

□

The following property allows to deduce from the weak $P(k,\varepsilon,\delta)$ property a stronger $P(k,\varepsilon)$ property when $P(k,\varepsilon,\delta)$ holds for the whole metric space.

Proposition 2.3.3. *If f is $P(k,\varepsilon,\delta)$ on $\mathbb{R}^m$, then f is also $P(k+\varepsilon/\delta,\varepsilon)$ on $\mathbb{R}^m$.*

Proof. Let x and y in $\mathbb{R}^m$. Let $n = \lfloor d(x,y)/\delta \rfloor$, we can apply the $P(k,\varepsilon,\delta)$ between the points x_i and x_{i+1} with $x_i = x + i\delta(y-x/d(x,y))$ for $i = 0$ to n. Note that $d(x_i,x_{i+1}) = \delta$ and $x_0 = x$. By using the hypothesis between x_i and x_{i+1} we get

$$d_n(f(x_i), f(x_{i+1})) \leq k d_m(x_i, x_{i+1}) + \varepsilon.$$

We derive

$$d_n(f(x_i), f(x_{i+1})) \leq (k + \frac{\varepsilon}{d(x_i,x_{i+1})}) d_m(x_i, x_{i+1}).$$

From the triangular inequality,

$$d_n(f(x_0), f(x_n)) \leq \sum_{i=0}^{n-1} d_n(f(x_i), f(x_{i+1}))$$

We derive

$$d_n(f(x_0), f(x_n)) \leq (k + \frac{\varepsilon}{\delta} \sum_{i=0}^{n-1} d_m(x_i, x_{i+1}).$$

Furthermore, from

$$d_m(x_0,x_n) = n\delta = \sum_{i=0}^{n-1} d_m(x_i,x_{i+1}).$$

We derive

$$d_n(f(x),f(x_n)) \leq (k+\frac{\varepsilon}{\delta})d_m(x,x_n)$$

Since $d_m(x_n,y) \leq \delta$, we have

$$d_n(f(x_n),f(y) \leq kd_m(x_n,y)+\varepsilon.$$

By triangular inequality, we derive

$$d_n(f(x),f(y)) \leq (k+\frac{\varepsilon}{\delta})d_m(x,x_n)+kd_m(x_n,y)+\varepsilon$$

Finally, observe that $d_m(x,y) = d_m(x,x_n)+d_m(x_n,y)$. We conclude that

$$d_n(f(x),f(y)) \leq (k+\frac{\varepsilon}{\delta})d_m(x,y)+\varepsilon$$

□

2.3.2 Robustness with respect to the exact semantics

As we explained, properties about the function itself do not measure the deviation with respect to the exact semantics. To express this deviation, we can consider that the two functions f and f' are close if the distance d_∞ (definition 2.2.6) between them is less than some ε:

$$\forall x \in \mathbb{R}^m, d(f(x),f'(x)) \leq \varepsilon.$$

However, this definition has the disadvantage of not being compositional. Indeed, consider the two following function f and g on $\mathbb{R}$:

$$f(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0 \end{cases}$$

$$g(x) = x$$

Depending on the finite-precision semantics of the architecture that computes f and g, we can have $f' = f$ while $g' \neq g$. In particular, we could have $g'(0) = -\varepsilon$. In this scenario, we have $(f \circ g)(0) = 1$ while $(f' \circ g')(0) = 0$: the difference is not small anymore.

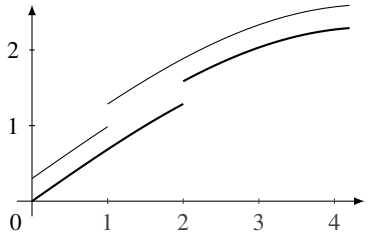

Figure 2.4: An example of functions that are (k,ε)-close

In addition, we also need to know how sensitive is f' to input errors otherwise, even if $f = f'$, the output error can be large.

To solve the compositionality problem and to keep information about the regularity of f' itself, we define a stronger property that we call (k,ε)-closeness property:

Definition 2.3.3 ((k,ε)-Closeness property)**.** *Let $(\mathbb{R}^m, d_m)$ and $(\mathbb{R}^n, d_n)$ be metric spaces. Let f and g be two functions from $\mathbb{R}^m$ to $\mathbb{R}^n$ and let $k, \varepsilon \in \mathbb{R}^+$. We say that g is (k,ε)-close to f if the following holds:*

$$\forall x, y \in \mathbb{R}^m, d_n(f(x), g(y)) \leq k d_m(x,y) + \varepsilon$$

With this definition, we ensure that if there was an error on the input the output error would still be bounded in an affine way. In particular, if there is no error on the input ($x = y$), f and g are within ε of one another.

Properties of our notion of closeness

We provide here the main interesting properties about this definition. At first, we have a compositional property about $f \circ g$.

Proposition 2.3.4. *If f and f' are (k,ε)-close and g and g' are (k',ε')-close then $f \circ g$ and $f' \circ g'$ are $(kk', \varepsilon + k\varepsilon')$-close.*

Proof. Let x and y in $\mathbb{R}^m$, we have

$$d_q(f(g(x)), f'(g'(y))) \leq k d_n(g(x), g'(y)) + \varepsilon$$

Then, we derive:

$$d_q(f(g(x)), f'(g'(y))) \leq k(k' d_m(x,y) + \varepsilon') + \varepsilon$$

□

The following straightforward observation provides a link between the notion of (k,ε)-closeness and the property $P(k,\varepsilon)$ defined previously.

Remark 1. *f is $P(k,\varepsilon)$ if and only if f is (k,ε)-close to itself.*

The following proposition relates the property of being (k,ε)-close and the one of being k-Lipschitz.

Proposition 2.3.5. *If f is k-Lipschitz and $\|f-f'\|_\infty \leq \varepsilon$ then f and f' are (k,ε)-close.*

Proof. Since k is k-Lipschitz, we have:

$$d_n(f(x),f(y)) \leq k d_m(x,y)$$

Since $\|n-n'\|_\infty \leq \varepsilon$,

$$d_n(f(y),f'(y)) \leq \varepsilon$$

We conclude by applying the triangular inequality:

$$d_n(f(x),f'(y)) \leq k d_m(x,y) + \varepsilon$$

□

Finally, we should notice that not all functions are (k,ε)-close to themselves. In fact, some function are not (k,ε)-close to any function.

Theorem 2.3.1. *If there exist u,v, $d(f(u),f(v)) > kd(u,v)+2\varepsilon$ then there exist no function g such that f and g are (k,ε)-close.*

Proof. Let u, v such that $d(f(u),f(v)) > kd(u,v)+2\varepsilon$.

Assume, by contradiction, that there exists g such that f and g are (k,ε)-close.

From the definition of closeness, we get

$$d(f(u),f'(u)) \leq \varepsilon$$

and

$$d(f'(u),f(v)) \leq kd(u,v)+\varepsilon.$$

Hence, by applying the triangular inequality, we derive

$$d(f(u),f(v)) \leq kd(u,v)+2\varepsilon.$$

Which contradicts the assumption of the theorem. □

2.4 Conclusion

In this chapter, we have presented the finite-precision representation and we explained that, even if for one operation the rounding error is as precise as the rounding of the exact operation, it is not the case anymore for a sequence of multiple instructions.

Then we presented the main analysis methods for dealing with this problem. There are mainly two issues when analyzing a code: the propagation of existing errors and the internal errors added by the program itself. We have proposed general definitions based on distances since our main purpose is not to try to get the best possible bound to the error, but to propose principles to prove robustness and safety properties.

$\star$

CHAPTER 3

DIFFERENTIAL PRIVACY

Contents

3.1 Introduction

In the last section, we provided tools to measure the computational error in a way independent of the given architecture. The notions of (k, δ)-closeness and the property $P(k, \varepsilon)$ allow to provide a bound to the maximal deviation from the exact semantics and from the initial inputs errors, respectively. Such kind of bounds are useful for program analysts since they allow measuring a

loss of precision. Once we know a program to be (k,ε)-close to its exact semantics, it is possible to know how many bits are not significant because the error amplitude is greater. Then, either the precision is enough and the program is certified correct or it is not sufficient for its use and the program has to be designed again, using data types whose implementation guarrantees more precision.

The traditional methods to analyze the error usually consider an environment where all parts are trusted. This is often the case when the program processes information internally to some system, like an air plane controller. In this chapter, we investigate another context where results are not used internally but transmitted for further processing.

Here we are in a security and confidentiality setting. Indeed, if an agent provides the result of a computation this does not mean that it wants to reveal the data from which it computes the result: some input data may be secret even if the result can be revealed. We investigate here the security issues caused by the finite precision. More specifically, we study the information leak about the input data that can be caused by the errors in the result, assuming that the adversary knows the result and the way the program works (because the program may be public for example). The standard techniques to measure the security breach do not apply, because those techniques analyze of the system in the *ideal* (i.e. *exact*) semantics and do not reveal the information leaks caused by the implementation.

Consider, for instance, the following simple program

$$\text{if } f(h) > 0 \text{ then } \ell = 0 \text{ else } \ell = 1$$

where h is a high (i.e., confidential) variable and ℓ is a low (i.e., public) variable. Assume that h can take two values, v_1 and v_2, and that both $f(v_1)$ and $f(v_2)$ are strictly positive. Then, in the ideal semantics, the program is perfectly secure, i.e. it does not leak any information. However, in the implementation, it could be the case that the test succeeds in the case of v_1 but not in the case of v_2 because, for instance, the value of $f(v_2)$ is below the smallest representable positive number. Hence, we would have a total disclosure of the secret value.

The example above is elementary but it should give an idea of the pervasive nature of the problem, which can have an impact in any confidentiality setting, and should therefore receive attention by those researchers interested in (quantitative) information flow.

From this example, we see that errors due to finite precision can provide additional secret information. This is due to the fact that errors should not be considered as non deterministic for all agents but controlled by the attacker. From the attacker point of view errors are knowledge.

In this chapter, we consider a more tricky case where the agent returns a noisy answer by using random numbers. Often the addition of random noise is done in purpose to prevent access

to secret values. We will see that, however, since noises are generated in finite-precision, these noises contain computational-error perturbations that allow an attacker to retrieve secrets.

To be able to have concretes examples and already existing specifications, we will study the particular case of *differential privacy*.

Differential privacy is an approach to the protection of private information in the field of statistical databases. Statistical databases are databases containing individual records, and aim at providing general information like finding plausible causes for diseases, social laws, or tendencies about politic opinion. In most countries, statistical databases are legal only if the anonymity and privacy of participants are preserved. Moreover, it is likely that people will not participate to a survey if they know that their personal information will be revealed to anybody. As we explain further in section 3.3, granting anonymity and privacy is not a trivial task.

Differential privacy has been first proposed in [Dwo06, DMNS06] as a formal approach to preserve the anonymity an privacy of the participants in a statistical database. This approach is now being used in many other domains ranging from programming languages [BKOB12, GHH+13] to social networks [NS09] and geolocation [MKA+08, HR11, ABCP13].

The key idea behind differential privacy is that whenever someone queries a dataset, the reported answer should not allow him to distinguish whether a certain individual record is in the dataset or not. More precisely, the presence or absence of the record should not change significantly the probability of obtaining a given answer. The standard way of achieving such a property is by using an *oblivious mechanism*[1] which consists in adding some noise to the true answer. Now the point is that, even if such a mechanism is proved to provide the desired property in the ideal semantics, its implementation may induce errors that alter the least significant digits of the reported answer and cause significant privacy breaches. Let us illustrate the problem with an example.

Example 3.1.1. Consider the simplest representation of reals: the fixed-point numbers defined in subsection 2.1.1. Each value is stored in a memory cell of fixed length. In such cells, the last d digits represent the fractional part. Thus, if the value (interpreted as an integer) stored in the cell is z, its semantics (i.e., the true real number being represented) is $z \cdot 2^{-d}$.

To grant differential privacy, the standard technique consists in returning a random value with probability $p(x) = 1/2b \cdot e^{-|x-r|/b}$ where r is the true result and b is a scale parameter which depends on the degree of privacy to be obtained and on the sensitivity of the query which is the maximal difference in the result when one entry is removed or added. To get a random variable with any specific distribution, in general, we need to start with an initial random variable provided by a primitive of the machine with a given distribution. To simplify the example, we

[1] The name "oblivious" comes from the fact that the final answer depends only on the answer to the query and not on the dataset.

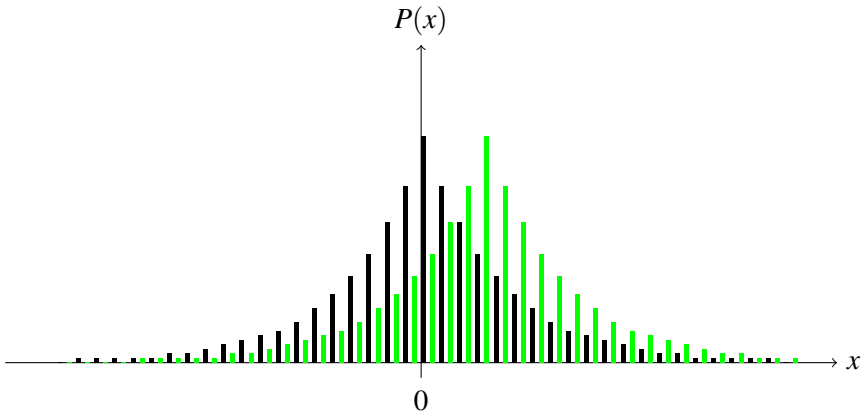

Figure 3.1: The probability distribution of the reported answers after the addition of Laplacian noise $P_{bX}(x)$ with $b = 4$, for the true answer $r_1 = 0$ (black) and $r_2 = 3 \cdot 2^{-4} + 2^{-5}$ (green).

assume that the machine already provides a Laplacian random variable X with a scale parameter 1. The probability distribution of such an X is $p_X(x) = 1/2 e^{-|x|}$. Hence, if we want to generate the random variable bX with probability distribution

$$p_{bX}(x) = \frac{1}{2b} \cdot e^{-\frac{|x|}{b}},$$

we can just multiply by b the value $x = z \cdot 2^{-d}$ returned by the primitive.

Assume that we want to add noise with a scale parameter $b = 2^n$ for some fixed integer n (b can be big when the sensitivity of the query and the required privacy degree are high). In this case, the multiplication by 2^n returns a number $2^n z \cdot 2^{-d}$ that, in the fixed-point representation, terminates with n zeroes. Hence, when we add this noise to the true result, we return a value whose representation has the same n last digits as the secret. For example, assume $b = 2^2 = 4$ and $d = 6$. Consider that the true answers are $r_1 = 0$ and $r_2 = 1 + 2^{-5}$. In the fixed-point representation, the last two digits of r_1 are 00, and the last two digits of r_2 are 10. Hence, even after we add the noise, it is still possible to determine which was the true value between r_1 or r_2. Note that the same example holds for every $b = 2^n$ and every pair of true values r_1 and r_2 which differ by $(2^n k+h)/2^d$ where k is any integer and h is any integer between 1 and $2^n - 1$. Figure 3.1 illustrates the situation for $n = 2$, $b = 4$, $d = 6$, $k = 3$ and $h = 2$.

Another attack, based on the IEEE standard floating-point representation [IEE08], was presented in [Mir12]. In contrast to [Mir12], we have chosen an example based on the fixed point representation because it allows to illustrate more distinctively a problem for privacy which rises

from the finite precision[2] and which is, therefore, pandemic. This is not the case for the example in [Mir12]: fixed-point and integer-valued algorithms are immune to that attack.

In this chapter, we propose a solution to fix the privacy breach induced by the finite-precision implementation of a differentially-private mechanism for any kind of implementation. Our main concern is to establish a bound on the degradation of privacy induced by both the finite representation and by the computational errors in the generation of the noise. In order to achieve this goal, we use the concept of *closeness* introduced in chapter 2, which allows us to reason about the approximation errors and their accumulation. In addition, we make as few assumptions as possible about the procedure for generating the noise. In particular, we do not assume that the noise has a linear Laplacian distribution: it can be any noise that provides differential privacy and whose implementation satisfies a few properties (normally granted by the implementation of real numbers) which ensure its closeness. We illustrate our method with two examples: the classic case of the univariate (i.e., linear) Laplacian, and the case of the bivariate Laplacian. The latter distribution is used, for instance, to generate noise in privacy-aware geolocation mechanisms [ABCP13].

3.1.1 Related work

As far as we know, the only other work that has considered the problem introduced by the finite precision in the implementation of differential privacy is [Mir12]. As already mentioned, that paper showed an attack on the Laplacian-based implementation of differential privacy within the IEEE standard floating-point representation[3]. To thwart such an attack, the author of [Mir12] proposed a method that avoids using the standard uniform random generator for floating point (because it does not draw all representable numbers but only multiple of 2^{-52}). Instead, his method generates two integers, one for the mantissa and one for the exponent in such a way that every representable number is drawn with its correct probability. Then it computes the linear Laplacian using a logarithm implementation (assumed to be full-precision), and finally it uses a snapping mechanism consisting in truncating large values and then rounding the final result.

The novelties of our work, w.r.t. [Mir12], consist in the fact that we deal with a general kind of noise, not necessarily the linear Laplacian, and with any kind of implementation of real numbers, not necessarily the IEEE floating point standard. Furthermore, our kind of analysis allows us to measure how safe an existing solution can be and what to do if the requirements needed for the safety of this solution are not met. Finally, we consider our correct implementa-

[2] More precisely, the problem is caused by scaling a finite set of randomly generated numbers. It is easy to prove that the problem raises for any implementation of numbers, although it may not raise *for every point* like in the case of the fixed-point representation.

[3] We discovered our attack independently, but [Mir12] was published first.

tion of the bivariate Laplacian also as a valuable contribution, given its practical usefulness for location-based applications.

The only other work we are aware of, considering both computational error and differential privacy, is [CGLN11] However, that paper does not consider at all the problem of the loss of privacy due to implementation error: rather, they develop a technique to establish a bound on the error, and show that this technique can also be used to compute the sensitivity of a query, which is a parameter of the Laplacian noise.

3.1.2 Plan of the chapter

This chapter is organized as follow. In section 3.2, we recall some mathematical definitions and introduce some notation. In section 3.3, we describe the standard Laplacian-based mechanism that provides differential privacy in a theoretical setting. In section 3.4, we discuss the errors due to the implementation, and we consider a set of assumptions which, if granted, allows us to establish a bound on the irregularities of the noise caused by the finite-precision implementation. Furthermore we propose a correction to the mechanism based on rounding and truncating the result. Section 3.5 contains our main theorem, stating that with our correction the implementation of the mechanism still preserves differential privacy, and establishing the precise degradation of the privacy parameter. The two sections that follow present some applications of our result: Section 3.6 illustrates the technique for the case of Laplacian noise in one dimension and section 3.7 shows how our theorem applies to the case of the Euclidean bivariate Laplacian. Section 3.8 concludes and discusses some future work.

3.2 Preliminaries and notation

In this section, we recall some basic mathematical definitions and we introduce some notation that will be useful in the rest of the chapter. We will assume that the queries give answers in $\mathbb{R}^m$. Examples of such queries are the tuples representing, for instance, the average height, weight, and age of the people in the database. Another example comes from geolocation, where the domain is $\mathbb{R}^2$.

3.2.1 Geometrical notations

Let $S \subseteq \mathbb{R}^m$.

We denote by S^c the *complement of* S, i.e., $S^c = \mathbb{R}^m \setminus S$.

The *diameter of* S is defined as

$$\varnothing(S) = \max_{x,y \in S} d(x,y).$$

For $\varepsilon \in \mathbb{R}^+$, the $+\varepsilon$-*neighbor* and the $-\varepsilon$-*neighbor* of S are defined as

$$S^{+\varepsilon} = \{x \mid \exists s \in S, d(x,s) \leq \varepsilon\}$$

$$S^{-\varepsilon} = \{x \mid \forall s \in \mathbb{R}^m, d(x,s) \leq \varepsilon \implies s \in S\}$$

These two definitions relate to each other with the following properties. The definitions coincide for $\varepsilon = 0$:

$$S^{+0} = S^{-0} = S$$

From one definition, we can define the other one through the complement of S:

Proposition 3.2.1. *We have the following equality:*

$$(S^{-\varepsilon})^c = (S^c)^{+\varepsilon}$$

Proof. By definition, the complement of a set defined by the points where a property holds is the set where the property does not hold, so we have:

$$(S^{-\varepsilon})^c = \{x \mid \neg(\forall s \in \mathbb{R}^m, d(x,s) \leq \varepsilon \implies s \in S)\}$$

This is equivalent to:

$$(S^{-\varepsilon})^c = \{x \mid \exists s \in \mathbb{R}^m, d(x,s) \leq \varepsilon \wedge s \notin S\}$$

This can be rewritten as:

$$(S^{-\varepsilon})^c = \{x \mid \exists s \in S^c, d(x,s) \leq \varepsilon\}$$

□

For $x \in \mathbb{R}^m$, the *translations* of S by x and $-x$ are defined as

$$S + x = \{y + x \mid y \in S\}$$

$$S - x = \{y - x \mid y \in S\}$$

where $+$ and $-$ are the vectorial spaces operations presented in section 2.2.3.

3.2.2 Measure theory

In this chapter, we will make use of measure theory. We recall, therefore, the basic notions of measure theory and some of its properties.

Definition 3.2.1 (σ-algebra and measurable space)**.** *A σ-algebra $\mathcal{T}$ for a set M is a nonempty set of subsets of M that is closed under complementation (wrt to M) and (potentially empty) enumerable union. The tuple $(M,\mathcal{T})$ is called a* measurable space*.*

Definition 3.2.2 (Measure)**.** *A* positive measure *μ on a measurable space $(M,\mathcal{T})$ is a function $\mu : \mathcal{T} \to \mathbb{R}^+ \cup \{0\}$ such that $\mu(\emptyset) = 0$, and if (S_i) is a enumerable family of disjoint subsets of M then*

$$\sum \mu(S_i) = \mu(\bigcup S_i).$$

A positive measure μ where $\mu(X) = 1$ is called a probability measure*.*

We will make use of the Lebesgue measure λ on $(\mathbb{R}^m, \mathcal{S})$ where $\mathcal{S}$ is the Lebesgue σ-algebra. The Lebesgue measure is the standard way of assigning a measure to subsets of $\mathbb{R}^m$.

Definition 3.2.3 (Measurable function)**.** *Let $(M,\mathcal{T})$ and (V,Σ) be two measurable spaces. A function $f : M \to V$ is measurable if $f^{-1}(v) \in \mathcal{T}$ for all $v \in \Sigma$.*

Definition 3.2.4 (Absolutely continuous)**.** *A measure ν is absolutely continuous according to a measure μ, if for all $M \in \mathcal{S}$, $\mu(M) = 0$ implies $\nu(M) = 0$.*

Theorem 3.2.1 (Radon-Nikodym)**.** *If a measure is absolutely continuous according to the Lebesgue measure then we can express it as an integration of a function f called the* density function *of the measure:*

$$\mu(M) = \int_M f(x)\, d\lambda$$

Definition 3.2.5 (Support)**.** *Let μ be a measure on $(\mathbb{R}^m, \mathcal{S})$, then the support of μ is defined to be the set of all points $x \in X$ for which every ball B_x with a positive radius that contains x has positive measure:*

$$\operatorname{supp}(\mu) := \{x \in X | x \in B_x \in T \implies \mu(B_x) > 0\}$$

Definition 3.2.6 (Essential supremum)**.** *The essential supremum of a function f,* $\operatorname{ess\,sup} f$*, is defined by*

$$\operatorname{ess\,sup} f = \inf\{a \in \mathbb{R} : \mu(\{x : f(x) > a\}) = 0\}$$

3.2.3 Probability theory

We focus now on the particular case of measure theory which is probability theory, and its specific definitions.

Definition 3.2.7 (Probability space)**.** *A probability space consists of three parts:*

- *A sample space, Ω, which is the set of all possible outcomes.*
- *A set of events $\mathcal{F}$, where each event is a set containing zero or more outcomes. $\mathcal{F}$ must be a σ-algebra on Ω.*
- *The assignment of probabilities to the events, that is, a measure P on $(\Omega, \mathcal{F})$ such that $P(\Omega) = 1$. .*

Definition 3.2.8 (Random variable)**.** *Let $(\Omega, \mathcal{F}, P)$ be a probability space and $(E, \mathcal{E})$ a measurable space. Then a random variable is a measurable function $X : \Omega \to E$. We shall use the expression $P[X \in B]$ to denote $P\left(X^{-1}(B)\right)$.*

Let $f : \mathbb{R}^m \to \mathbb{R}^m$ be a measurable function and let $X : \Omega \to \mathbb{R}^m$ be a random variable. In this paper, we will use the notation $f(X)$ to denote the random variable $f(X) : \Omega \to \mathbb{R}^m$ such that $f(X)(\omega) = f(X(\omega))$. In particular, for $m \in \mathbb{R}^m$ we denote by $m+X$ the random variable $m+X : \Omega \to \mathbb{R}^m$ such that $(m+X)(\omega) = X(\omega) + m$.

Definition 3.2.9 (Density function)**.** *Let $X : \Omega \to E$ be a random variable. If there exists a function f such that, for all $S \in \mathcal{S}$, $P[X \in S] = \int_S f(u)\,du$, then f is called the* density function *of X.*

In this chapter, we use the following general definition of the Laplace distribution (centered at zero).

Definition 3.2.10 (Laplace distribution)**.** *The density function F of a Laplace distribution with scale parameter b is $F_b(x) = K(b)e^{-b\|x\|}$ where $K(b)$ is a normalization factor which is determined by imposing $\int_S F_b(x)\,dx = 1$.*

This definition is not the standard definition of the multivariate Laplacian considered, for instance, in [EKL06]. Indeed, the standard definition generalizes the characteristic function of the Laplace distribution while, here, we generalize the density function. The reason to use this definition of Laplacian instead of the standard one is that this is the only one that ensures differential privacy (in the exact semantics).

Definition 3.2.11 (Cumulative function)**.** *The cumulative distribution function of a real-valued random variable X is the function given by*

$$F_X(x) = P(X \leq x),$$

Definition 3.2.12 (Joint probability)**.** *Let (X,Y) be a pair of random variable on $\mathbb{R}^m$. The joint probability on (X,Y) is defined for all $I,J \in \mathcal{S}$ as:*

$$P[(X,Y) \in (I,J)] = P[X \in I \wedge Y \in J].$$

Definition 3.2.13 (Marginals)**.** *Let X and $Y : (\Omega, \mathcal{F}, P) \to (\mathbb{R}^m, \mathcal{S})$. The marginal probability of the random variable (X,Y) for X is defined as:*

$$P[X \in B] = \int_{\mathbb{R}^m} P[(X,Y) \in (B, \mathrm{d}y)].$$

3.3 Differential privacy

As we stated in the introduction of this chapter, differential privacy emerged from a general problem of confidentiality. In this section, we first present the historical background to show why this approach has been chosen and why other approaches are problematic. Then, we define formally the differential privacy property. Next, we explain how differential privacy can be used and we illustrate some of its properties. In the last subsection, we explain how this property is achieved in case of real valued answer.

3.3.1 Context and vocabulary

A statistical database has to be considered as a collection of rows. Each rows contains tuples of values that can be seen as a vector. In practice rows can contain any kind of type like strings for names, integers for id numbers, booleans for answer to yes/ no question or reals for measures like height. Here, for simplicity, we only consider reals which are the kind of data for which the problem of finite precision arises.

We consider three agents concerned by the database: the participants, also called respondents, that provide personal data to the database, the curator of the database, and the analysts which ask for queries.

In our setting, we consider that the owner can be trusted by the respondents while the analysts are not trusted by the respondent. This is the case when the database is own by a public institution for instance. The analysts can be researchers looking for correlations between some diseases or

companies that are looking for new tendencies. In another context, the database can consist of the employee of a company that has all the private information and wants to mandate an external analyst to adapt its strategy, but does not want to reveal any personal information.

In these situations, the analyst chooses the queries to ask to the database. He can ask simple queries like "how many people in the database have diabetes?", or more complex queries. In general, we assume no limitations in the expressiveness of the query language, i.e., we assume that it is Turing complete. Since the query language is Turing complete, it is undecidable whether two syntactically different queries are equivalent.

Since any kind of query can be asked, the analyst can ask for "What is the average age of people named X". Answering to such a query would of course break the privacy of X. To avoid the leakage an agent called the curator "sanitizes" the answer.In the next subsection, we present some of the approach that have been proposed in the literature, and we show that are not satisfactory.

3.3.2 Previous approaches to the privacy problem

Before we introduce the formal definition of differential privacy we present the previous proposed methods to grant privacy and we discuss their shortcomings.

The first approaches to sanitize private data consisted in releasing an altered database and the analyst was able to answer himself to his queries on this altered database.

Anonymization The first approach to prevent leakage was to remove all identifiers of individuals. Mainly, the identifiers (name, address, id number, etc.) were removed from the survey while all other information was provided to the analyst. Such a mechanism is not safe however. Indeed, records contains other information like gender, ZIP code, birthday or whatever is publicly available. Even if each of these data separately does not allow to identify an individual, some combination of them might. For this reason, they are called quasi-identifiers.

***k*-anonymity** The k-anonymity property [Swe02] has been coined to prevent the previous attack. The k-anonymity requirement consists in providing an altered database such that whatever is a query, it is not possible to link the result to less than k rows. The underlying idea is that, since there are always at least k individuals that are bunched with one specific individual, so that it is not possible to extract the row corresponding to a particular participant. To achieve k-anonymity, some fields are either removed or generalized. For instance, in a survey, the ZIP code can be removed while the birth date can be generalized such that only the year appears.

This solution, however, has weaknesses. First, since there can be several databases that are

protected independently, it is possible that released data from one curator grants privacy for its own database but it allows to attack another database thanks to the additional correlations it provides. Next, getting k rows with the same public information does not means that the k rows are different according to the private data. For instance, if there are k individual with the same birthday and that birthday is the only public information, the database is k-anonymous. However, if, by chance, all people born at some date have more or less the same incomes then it is possible to have a precise information about the incomes of all the individuals having this birthday.

To solve this problem, other improvements have been tried like l-diversity [MKGV07] that also requires for the k rows to have different private values. This approach is too restrictive however when considering binary private data like having some illness where 99.99% of participants are negative. The t-closeness property [LL07] amends the problem by asking that the distribution in the sampling is close to the distribution of values in the whole database.

Noise perturbations To provide the t-closeness property for a database, one proposal was based on the use of noise perturbations [RMFDF08]. Noise perturbations consist in modifying the initial database by adding noise to all values. In the family of noise perturbations techniques, a simple procedure consists in replacing some percentage of the values by random values so that when an answer is returned about a single row it is not possible to determine if the result corresponds to the real answer or to a random value. Such kind of techniques can be sufficient for simple data but if data are too complex it either becomes unfeasible or the alteration is too invasive and the altered database provide a poor utility. For instance, it has been shown that graph relationship can be re-identified even with addition of fake links [NS09].

Towards differential privacy To avoid the problems above, a different approach was proposed. Instead of altering the database and then releasing it to the analyst, the curator keeps the database secret and just communicates answers to the queries of the analyst. In this setting, it is possible to have a better control of the data leakage.

The basic principle consists in providing a noisy answer to each query. However, if the same query is asked several times then the analyst could average the results so to get more and more close to the true answer. It is also not possible to always add the same noise to the same query since two queries can be syntactically different while they are semantically identical, and semantical identity is undecidable.

In any case, we need a formal definition of what we consider to be a good privacy guarantee. An idea for the definition might be that the knowledge about any particular individual does not change much by answering a query. Such kind of definition would be too strict however.

Indeed, since we want to provide general information, it might be possible that the analyst already has a particular knowledge about some individual which links a general information to one of this individual. For instance, if we know that the income of someone is exactly the same as the average then, revealing the average income of the people in the database reveals the exact income of this individual. However, we note that this problem arises independently from whether or not this individual participates to the database. Now, one of the main issues is to reassure the individuals that their participation in the database is safe,i.e., it does not harm their privacy. For this reason, the definition of differential privacy is based on the idea that whether an individual participates or not in the database does not affect much the probabilistic knowledge.

3.3.3 Definition of differential privacy

We start by formalizing the notion of participation versus non-participation of an individual in the database.

Definition 3.3.1 (adjacent databases)**.** *Given two databases D_1 and D_2, we denote by $D_1 \sim D_2$ the fact that D_1 and D_2 differ by exactly one row. Namely, D_2 is obtained from D_1 by adding or removing the data of one individual.*

In order to ensure (almost) the same knowledge whether an individuals is in the database or not, it is not a good idea to use a deterministic noise like rounding the result. Indeed, such a method leaks information each time one individual causes the answer to shift to another rounding. For instance, if the value of some attribute for each individual is between 0 and 0.01, and we decide to round the result to the closest integer, then two successive sum queries (one on a subset S and another one on S plus one individual i) can provide information any time there is a shift one integers. If we assume that the analyst has a side information which is the true value of S, like 9.995, then he can learn whether the value of i is greater or smaller than 0.005.

To avoid this problem, the solution consists on adding a random noise. In this way, even if the analyst already knows the whole data but one individual, the probabilistic noise still hides the personal result. We define a noisy answer as follow.

Definition 3.3.2 (randomized mechanism)**.** *A randomized mechanism $\mathcal{A}$ is a function that takes a database D and a query q and returns a random variable X.*

$$X = \mathcal{A}_q(D)$$

We omit the q parameter when there is no ambiguity.

We can now provide the formal definition of differential privacy. We denote by $\mathcal{D}$ the set of all possible databases, which, in general, may be an infinite set since the number of entries may not be bounded.

Definition 3.3.3 (ε-differential privacy)**.** *A randomized mechanism $\mathcal{A} : \mathcal{D} \to \mathbb{R}^m$ is ε-differentially private if for all databases D_1 and D_2 in $\mathcal{D}$ with $D_1 \sim D_2$, and all $S \in \mathcal{S}$ (the Lebesgue σ-algebra on $\mathbb{R}^m$), we have :*

$$P[\mathcal{A}(D_1) \in S] \leq e^{\varepsilon} P[\mathcal{A}(D_2) \in S]$$

This definition is based on the idea that, since the probabilities of obtaining a certain answer are similar, it is not possible to extract probabilistic information from the presence of an individual.

3.3.4 Some properties of differential privacy

We now mention some of the main results about differential privacy in order to show the strength of this definition.

Since differential privacy is a proportional relation, there are two immediate propositions about compositions.

Proposition 3.3.1 (Several users privacy [Dwo11])**.** *A ε-differentially private mechanism preserve privacy for a single user. If we are interested to protect groups on n users, the definition of adjacent database should be that D_1 and D_2 differs for the addition or removal of n users. A simple transitivity argument leads to conclude that according to this definition the same mechanism is nε-differentially private.*

Proposition 3.3.2 (Multiple queries)**.** *If n successive answers are returned such that the i-th answer is issued by a ε_i-differentially private mechanism, then the global mechanism is $\sum_1^n \varepsilon_i$-differentially private.*

As a corollary, a standard protocol consists to attribute a privacy budget to each analyst. In this protocol, the analyst chooses how much of the privacy budget he wants to spend for a query and the curator returns the answer with a noise corresponding to this amount. When the budget is over, the analyst cannot ask further queries. By the above proposition, this protocol grants ε-differential privacy.

A final proposition is about compositionality of mechanisms which states that once a result enjoy differential privacy then any post process is also differentially private.

Proposition 3.3.3. *If a mechanism $\mathcal{A}$ is ε-differentially private in range $\mathbb{R}^m$, then for any function $f : \mathbb{R}^m \to \mathbb{R}^m$, we have that $f \circ \mathcal{A}$ is also ε-differentially private.*

For instance, if the possible domain for the true value is bounded, then, when the returned answer is outside this domain, it is possible to return instead the closest value inside this domain without any loss of privacy.

3.3.5 Standard technique to implement differential privacy

Differential privacy is also interesting because it is a property easy to implement property, at least from a mathematical point of view.

There are several ways to achieve it. For instance, a possible solution consists in replacing some of the entries, randomly selected, by a random one. In the simplest case where answer have value in $\{0,1\}$, and where one half of the entries are replaced by a random one with equiprobability then it has been proved in [Dwo11] that such a mechanism is $\ln(3)$-differentially private. Such a mechanism, which enjoys also that the database owner has not to be trusted (since the respondent can provide a random answer), is an example of non oblivious mechanism.

Here we will be interested in the oblivious mechanisms, defined formally as follow.

Definition 3.3.4 (oblivious mechanisms)**.** *A process to sanitize the database is oblivious if it takes as input only the true answer, i.e., not the database. We use the notation $f_q(D)$ to denote the true answer on the database D and $f(D)$ when there is no ambiguity about q.*

Such class of mechanisms does not need to consider the database anymore, the only interesting parameter is the maximal deviation that one individual can generate for a given query. This leads to the following definition.

Definition 3.3.5 (sensitivity)**.** *The sensitivity Δ_f of a function $f : \mathcal{D} \to \mathbb{R}^m$ is*

$$\Delta_f = \sup_{D_1, D_2 \in \mathcal{D}, D_1 \sim D_2} d(f(D_1), f(D_2)).$$

In this class of oblivious mechanisms, we are interested in the particular subcase of mechanisms that only add a random value to the result.

Mechanism 1.

$$\mathcal{A}_0(D) = f(D) + X$$

These mechanisms are useful enough (in the exact semantics) and we will limit our study to them.Utility is a formal notion even though there is no unique definition, and it is an important notion. For instance, a mechanism that only returns a random variable is 0-differential private but since it never allows to learn anything, this mechanism is perfectly useless. For the subset of the additive mechanism the utility can be defined as a function of the variance between the true answer and the real value.

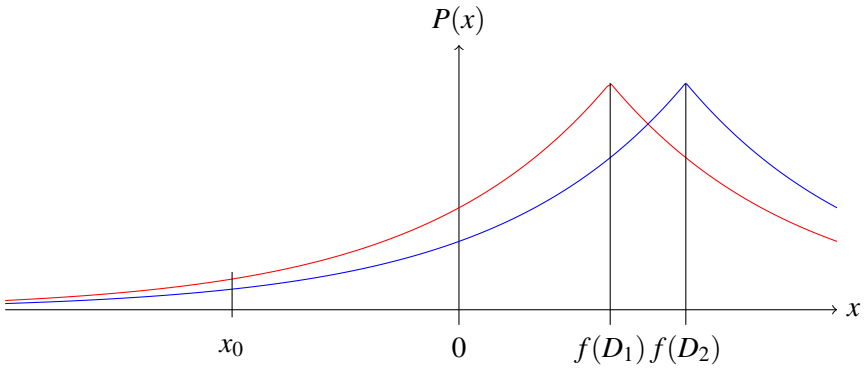

Figure 3.2: The two distributions corresponding to a true answer $f(D_1)$ and $f(D_2)$

In case of no prior knowledge, the Laplace distribution is an optimal distribution in term of ratio between utility and privacy for the class of additive noise (mechanism 1). Indeed, the Laplace enjoys a constant ratio of ε between two adjacent databases as illustrated in 3.2.

3.4 Error due to the implementation of the noise

In this section, we detail our quantification of the error due to the finite precision semantics. From the analysis of the implementation, we can extract properties of the chapter 2. We start by considering the different parts where finite precision causes deviation. In section 3.4.1, we briefly explain why we do not focus on the implementation of the function computing the true answer. In section 3.4.2, we explain what is the generally used pattern to implement a random noise. This generation is made in two steps: generation of a uniformly distributed noise and then the computation of a function from this initial noise. In section 3.4.3, we detail this implementation and we provide a model that provide a bound on the maximal error. In section 3.4.4, we analyze how the function that transform the initial noise produces more deviation.

The rest of this section aims at using this maximal deviation to quantify the increase of the differential privacy parameter (how much leakage the finite precision adds to the theoretical leakage). In fact, the direct implementation of the algorithm in exact semantics without any adaptation is unsafe. So, it is not possible to get a bound on the leakage without any change in the algorithm. Therefore, during the quantification of the leakage, we introduce changes in the algorithm motivated both by the fact a proof is not possible without these changes and some illustrations on how the exact mechanism can be attacked. In section 3.4.5, we aims at proving that the total deviation is bounded. This is not the case in the initial mechanism, however. In order to get a maximal bound, we propose a safer mechanism that truncates the result. In

section 3.4.6, we consider a specific distance, the ∞-Wassertein distance, between distributions. We prove that the existence of a maximal bound of the deviation implies some bounds in the distance between the exact distribution and the finite-precision one. In section 3.4.7, we show that this bound is not precise enough and we motivate the need to round the result provided by our former mechanism. In section 3.4.8, we consider some problems caused by some specifics distribution and we measure an additional leakage due to the weakness of these distributions when they are implemented. At the end of this section, we are able to measure the leakage due to the finite precision. The measure itself and its proof constitute the next section.

3.4.1 The problem of the approximate computation of the true answer

Before any consideration about how the noise X is generated and induces errors when computed, we consider the errors appearing during the computation of f itself.

First of all, we note that, while X is produced by a unique algorithm, f is generated for each query from the syntax of the query. Indeed, X has to change according to the privacy budget we want to spend for the query and according to the sensitivity of the asked query. But these two parameters do not change the code of the function itself. In the case of the query, the user "codes" a function with the syntax of the language. Then the function is interpreted or compiled into the native language of the machine to the function f'.

If the function f' is not close to f, then the utility of the answer will be less valuable. In term of differential privacy however, it is not important that the function f' is used instead of f. The only possible leakage that may happen appears if the scale factor for the noise is based on Δ_f instead of $\Delta_{f'}$. For instance the following query :

$$f(D) = \min_{i \in D}(((i + \text{Big}) - \text{Big}) - i)$$

where Big is a huge number of the form 2^n should return the minimal value among all entries of an expression that is always 0 in the exact semantics. So if we compute Δ_f in the exact semantics we get 0. However, if the computation is done in the floating-point arithmetic with IEEE standard, $i + \text{Big} - \text{Big}$ is equal to 0 if $i < 2^{-52}\text{Big}$ else $i + \text{Big} - \text{Big}$ is greater than 0. If we use Δ_f then no noise will be added to the answer of f', hence by asking this query, it is possible to learn if there exists some value greater than 2^{-52}Big in the database without spending any privacy budget.

There exist several ways to compute the sensitivity of a query. One of these methods consists in doing a statical analysis of the generated function to find a parameter k for which f is k-Lipschitz (such a mechanism has been proposed in [CGLN11]). Once we know that f is k-Lipschitz and that each entry belongs to some domain of diameter d, then we know that the

sensitivity is at most kd. To avoid the problem illustrated previously, we have to be careful that the k factor be actually computed in the finite-precision semantics and not in the exact one.

For the following, for readability reasons, we only mention Δ_f but we mean Δ'_f.

3.4.2 General method to implement real valued random variables

Computers are deterministic machines. To implement probabilistic algorithms, operating systems provide a random seeds: it is a value in a register that is updated according to the most erratic behaviors that the computers can access: mouse moves, some microtime values, etc.

Since the random seed use is limited (a user makes a limited number of mouse moves during a period), an additional mechanism simulates pseudo-random numbers through erratic functions.

Example 3.4.1. *In the Java language, the following function provides the new random integer from the last one:*

```
synchronized protected int next(int bits) {
        seed = (seed * 0x5DEECE66DL + 0xBL) & ((1L << 48) - 1);
        return (int)(seed >>> (48 - bits));
}
```

This is a deterministic function made from simple arithmetic operators and bits shifts.

In both cases, however, random variables are initially set with the help of an external process. An algorithm accesses these values through primitive of the language. For instance in the C language, there exists a unique primitive `rand()` that returns an integer between 0 and some fixed value `RAND_MAX`. There exist many functions in Java, but all of them call the `next()` function above, i.e. every generators are made from the integers random generator.

In order to generate real-valued random variables, a random integer has to be cast into a double (or any finite-precision representation type for reals). The most straightforward mechanism to do it consists of dividing the random integer by the greatest value of the random generator. In that case, we obtain a random value between 0 and 1. The actual mechanism in the Java library is given in figure 3.4. The distribution is uniform in some weak sense: the pseudo-random generator generates once and only once all possible values before looping. Since the set of all the possible values has a mean of $1/2$, when the random generator is called enough time, the average of generated numbers is $1/2$.

In case of floating point numbers it is possible to build a better generator [Mir12]. Indeed, the last method just provides numbers that are multiple of $1/mr$. However representable numbers are not equally spaced since the exponent allows to represent very small numbers. So to have

```
synchronized public double nextGaussian() {
    if (haveNextNextGaussian) {
            haveNextNextGaussian = false;
            return nextNextGaussian;
    } else {
            double v1, v2, s;
            do {
                    v1 = 2 * nextDouble() - 1;
// between -1.0 and 1.0
                    v2 = 2 * nextDouble() - 1;
// between -1.0 and 1.0
                    s = v1 * v1 + v2 * v2;
            } while (s >= 1 || s == 0);
            double multiplier = Math.sqrt(-2 * Math.log(s)/s);
            nextNextGaussian = v2 * multiplier;
            haveNextNextGaussian = true;
            return v1 * multiplier;
    }
 }
```

Figure 3.3: The source code of the Gaussian generator in the Random library of Java

a better generator, we need to draw an integer to be the mantissa and another according to the exponential law to the exponent.

We have seen various ways to get a uniform random variable in $[0,1[$. From a perfectly uniform random variable generator, it is theoretically possible to build any kind of random variable in any domain $\mathbb{R}^m$. There exists two main techniques to get other distribution from this initial one. The first one is used for real valued random variable ($m = 1$). Indeed, to draw a value from a random variable X on $\mathbb{R}$ distributed according to the cumulative function $C : \mathbb{R} \rightarrow]0,1]$, it is sufficient to pick a value u from the uniform generator in $]0,1]$ and then return $C^{-1}(u)$. The second one uses already generated random variables and combines them into some arithmetic expression to get a new distribution. For instance, to have a Gaussian distribution, the Java random library have a function (Figure 3.3) that makes two calls to the uniform pseudo-random generator `nextDouble()` (Figure 3.4).

The code of figure 3.3 highlights that the number q of random numbers drawn to generate a random variable does not depend on the dimension m of its co-domain. The only condition is $q \geq m$ due to cardinality reasons (in the example $q = 2$ while $m = 1$).

Since, several uniform random values can be drawn to generate the noise, we do not consider the probability distribution of one draw but the one of several draws whose its support is $[0,1]^q$.

```
  public double nextDouble() {
       return (((long)next(26) << 27) + next(27))
           / (double)(1L << 53);
}
```

Figure 3.4:

This drawn variable U' is an approximation of a uniformly distributed variable on $[0,1]^q$.

3.4.3 Errors due to the initial random generator

Now we have explained how the generation of a random number works, we explicit the kind of errors that can happen and how to bound them.

There are mainly three reasons why a uniform random generator may induce errors, i.e. the random variables generated are not perfectly independent uniformly distributed in $[0,1]^q$.

Equiprobability The first issue is inherent to the principle of generating random values itself. Whether the random value is drawn by dices or by a computer, there is remains a bias in the equiprobability. This bias comes at least from the initial seed (real world events are never perfectly equipossible). Even if this bias is not really part of our main concern, it should not be forgotten.

Dependency The next bias is due to the dependence of returned results when we pick several random values to generate the noise. Indeed, as we explained, most of the generator implementations are pseudo generators. When a value is picked, the next one is generated as a hash function of the first one. This means that if we have N possibilities for one choice then we also have N possible pairs of successive random values instead of N^2. This is represented in Figure 3.5. In the first square, we have represented all the possible outputs if the two random values were really independent (here $N = 25$ and so there are 625 possibilities). In the second square, we have represented the outputs when the second random value is fully determined by the first one.

Depending on how erratic is the pseudo random generator, it may be possible to observe big area where there is no possible value. Even if these areas are hard to compute for an attacker some of them might be found and then exploited: the computation have to be done once for all attacks on the same architecture. To avoid this leakage, it can be decided to uniformly spread the possible outputs of the random generator on N^2 by splitting the bits of the initial integer such that there is $\sqrt[m]{N}$ possible values on each axis (third square in figure 3.5). This is not a good

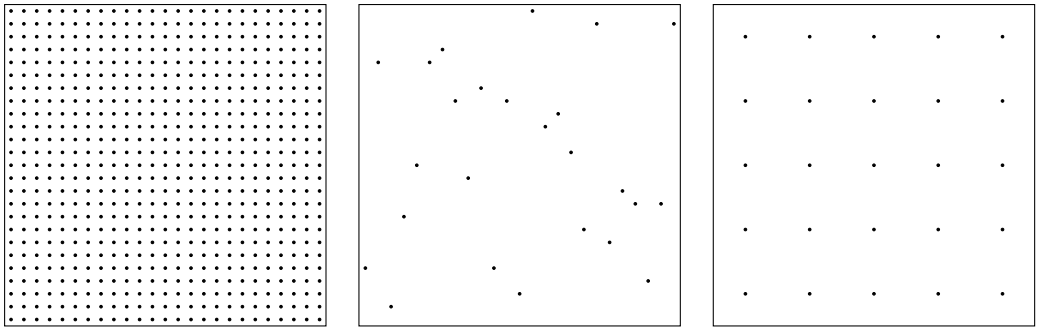

Figure 3.5: In the first picture is represented the intended possible values when picking two random numbers. In the second figure, the actual possible draws. The last one is the best repartition.

solution, however, since now the distribution of the possible value is perfectly known by any attacker without doing any computations. Moreover, this problem also appears in the case of multiple identical queries. A careful study has been made about this problem in [DLAMV12]. This problem is complex because it concerns the initial random-integer generator about its cryptographic properties. Here, we do not enter into the details of the problem. We only assume that the knowledge of this distribution needs too much time and space resources, so we do our modeling as if real valued random values were actually independent from each other.

Finite representation The most important bias, however, is due to the finite-precision representation itself. Indeed, whenever we use a perfect random generator, this generator has to return a result into the set of representable numbers. The following example shows that any finite implementation makes it impossible for a mechanism to achieve the degree of privacy predicted by the theory (i.e. the degree of privacy it has in the exact semantics). This example is more general than the one in the introduction in the sense that it does not rely on any particular implementation of the real numbers, just on the (obvious) assumption that in a physical machine the representation of numbers in memory is necessarily finite. On the other hand it is less "dramatic" than the one in the introduction, because it only shows that the theoretical degree of privacy degrades in the implementation, while the example in the introduction shows a case in which ε-differential privacy does not hold (in the implementation) for any ε.

Example 3.4.2. Consider the standard way to produce a random variable with a given probability law, such as the Laplace distribution. Randomness on most computers is generated with integers. When we call a function that returns a uniform random value on the representation of reals, the function generates a random integer z (with uniform law) between 0 and N (in practice $N \geq 2^{32}$) and returns $u = z/N$. From this uniform random generator, we compute $n(z/N)$

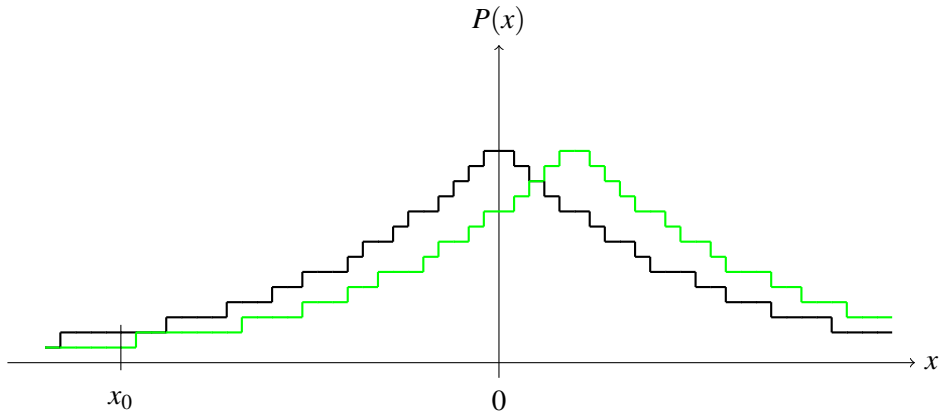

Figure 3.6: The probability distributions of Laplace noises generated from a discretized uniform generator

where n depends on the probability distribution we want to generate. For instance, to generate the Laplace distribution we have $n(u) = -b\,\mathrm{sgn}(u-1/2)\ln(1-2|u-1/2|)$ which is the inverse of the cumulative function of the Laplace distribution. However the computation of n is performed in the finite precision semantics, i.e. n is a function $\mathbb{F} \to \mathbb{F}$ where $\mathbb{F}$ is the finite set of the representable numbers. In this setting, the probability of getting some value x for our noise depends on the number of integers z such that $n(z/N) = x$: if there are k values for z such that $n(z/N) = x$ then the probability of getting x is k/N. This means, for instance, that, if the theoretical probability to draw x is $1.5/N$, then the closest probability that can be actually associated with this drawing of x is either $1/N$ or $2/N$ and in both cases the error is at least 33%. In figure 3.6, we illustrate how the error on the distribution breaks the differential-privacy ratio that holds for the theoretical distribution. The ratio between the two theoretical Laplacian distributions is $4/3$. However, since the actual distribution is issued from a discretization of the uniform generator, the resulting distribution is a step function. So on the domain where the theoretical probability is very low, like in x_0, the discretization creates an artificial ratio of 2 instead of $4/3$.

We model this last form of leakage in the following. We consider that a perfect random values $u = (u_1, \ldots, u_q) \in [0,1]^q$ is picked with all its decimals. Then we consider the random variable $U^{q\prime}$ actually provided as generated from a function $n_0 : \mathbb{R}^q \to \mathbb{R}^q$, $(u'_1, \ldots, u'_q) = n_0(u_1, \ldots, u_q)$ where n_0 is a mathematical function that takes u and returns the closest value u' in the set of the possible outputs of our pseudo random generator. For instance, if our pseudo random generator can only return numbers that are multiple of 2^{-53}, the n_0 function is the function that truncates numbers after the 53^{th} decimal bit. In figure 3.7, we illustrate such a shift: white circles represent the ideal values that should have been drawn while the black circles represents

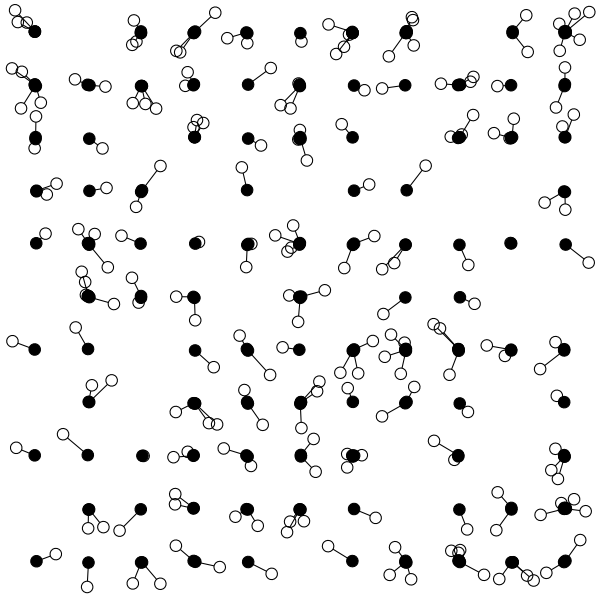

Figure 3.7: Generation of an ideal value (in white) and its finite-precision related value (in black)

the actual drawn value in the fixed-point representation. The size of the link between the two circles represents the initial error.

From this definition of n_0 we can define $\delta_0 \in \mathbb{R}^+$ that corresponds to the maximal distance between $n_0(u)$ and u for any $u \in [0,1]^q$. Denoting by Id the identity function, we have:

$$\delta_0 = \|n_0 - \mathrm{Id}\|_\infty. \tag{3.1}$$

So in case the random generator generates numbers that are multiple of δ, we have $\delta_0 = \delta$.

3.4.4 Errors due to the function transforming the noise

We have defined the error done by the initial uniform random generator, and we now consider the errors of computation when transforming this noise into a noise with the desired distribution. For instance, in the code 3.3, the function `nextGaussian` takes as an input a uniform random variable in $[0,1]^2$ (there are two calls to the uniform random variable `nextDouble`) and returns two independent variables with a Gaussian distribution.

Formally, given the random variable X chosen for the mechanism 1, there may exist a function n that takes a tuple $u \in [0,1]^q$ and returns a value in $\mathbb{R}^m$ such that the distribution of $n(U^q)$ (where U^q is the uniform random variable on $[0,1]^q$) is the same as the distribution of X. When such a function exists, it can be used to draw the variable X. Hence mechanism 1 can be specialized to:

Mechanism 2.

$$\mathcal{A}_0(D) = f(D) + n(U)$$

In other words, mechanism 2 specifies that we have an additive noise generated from a uniform one. Since we are considering errors of computations, we denote by n' the actual function in the finite-precision semantics and by X' the random variable actually generated. So we have in the exact semantics:

$$X = n(U)$$

and in the finite-precision semantics:

$$X' = n'(U') = n'(n_0(U))$$

In order to establish a bound on the difference between the probability distribution of X and X', we need some conditions on the implementation n' of n. For this purpose we use the notion of (k,δ)-closeness between two functions that we defined in Definition 2.3.3. Indeed, from equation 3.1, the fact Id is 1-Lipschitz and Proposition 2.3.5, n_0 and Id are $(1,\delta_0)$-close. Then, if n and n' are (k,δ)-close on $\mathbb{R}^m$, since n_0 and Id are $(1,\delta_0)$-close, from the weak compositionality property 2.3.4 , we get n and $n' \circ n_0$ to be $(k, k\delta_0 + \delta)$-close. We derive: $\forall u \in [0,1]^q, d((n' \circ n_0)(u), n(u)) \leq k\delta_0 + \delta$. Finally, if we denote by x a draw from the exact random variable and x' the corresponding draw in the finite precision, we have,

$$\forall x \in \mathbb{R}^m, d(x,x') \leq k\delta_0 + \delta$$

Unfortunately, we have the following impossibility property.

Proposition 3.4.1. *Any generation of random variable that achieves* ε*-differentially-privacy according to mechanism 2 cannot have an implementation which is* (k,δ)*-close to the exact semantics on* $[0,1]^q$*, for any k and* δ*.*

Remark 2. *The main idea is that, from Theorem 2.3.1, functions with a vertical asymptote are not close to any function.*

To prove this proposition, we need the following lemma which states that in a differentially private mechanism with additive noise there is no finite bound to the amplitude of the noise.

Lemma 3.4.1. *In any* ε*-differentially private mechanism 1 on a set of databases where at least two different true answers are possibles on two adjacent databases, the random variable X is such that* $\operatorname{ess\,sup}\|X\| = \infty$*.*

Proof. Let r_1 and r_2 two possible answers.

Let $M \in \mathcal{S}$ (the set of measurable sets) such that $\|M\|$ is bounded and $P(X \in M) > 0$. With the definitions about translations of section 3.2.1, we also have $P(r_1 + X \in M + r_1) > 0$, hence, since $\mathcal{A}$ is ε-differentially private, $P(r_2 + X \in M + r_1) > 0$ (otherwise the ratio between the two probabilities would be infinite). So we have

$$P(X \in M) > 0 \implies P(X \in M + r_1 - r_2) > 0.$$

Since, this is valid for any M, we have in particular, for any $h \in \mathbb{N}$:

$$P(X \in M + h(r_1 - r_2)) > 0 \implies P(X \in M + (h+1)(r_1 - r_2)) > 0.$$

From the assumption $P(X \in M) > 0$ and the last implication, we conclude, by induction, that for all $h \in \mathbb{N}$:

$$P(X \in M - r_1 + h(r_1 - r_2)) > 0 \tag{3.2}$$

Finally, since M is bounded, for any m there exists a h such that

$$\{0\}^m \cap M - r_1 + h(r_1 - r_2) = \emptyset \tag{3.3}$$

where $\{0\}^m$ is the ball centered in 0 of radius m. This means that the set $M - r_1 + h(r_1 - r_2)$ does not have any element e with $d(0, e) \leq m$. From equations 3.2 and 3.3, we have exhibited sets $(M - r_1 + h(r_1 - r_2))$ with non null probability that are arbitrary far from the origin. We can conclude that, for all $s \in \mathbb{R}$, $P(\|X\| > s) > 0$ i.e. $\operatorname{ess\,sup}\|X\| = \infty$. □

Proof of proposition 3.4.1. From Lemma 3.4.1, it follows that

$$\|n([0,1]^q)\|_\infty = \infty.$$

Moreover, $\|n([0,1]^q)\|_\infty = \infty$ and the fact that $[0,1]^q$ is bounded implies that, for all k and δ, there exist u, v, such that

$$d(n(u), n(v)) > k d(u, v) + 2\delta.$$

We conclude from Theorem 2.3.1 that n' (k, δ)-close to the exact semantics cannot exist. □

Figure 3.8 illustrates the problem: while the returned result is close enough to zero, the n function is safe enough. But then, the greater is the returned value, the higher is the error between the exact and the finite-precision semantics.

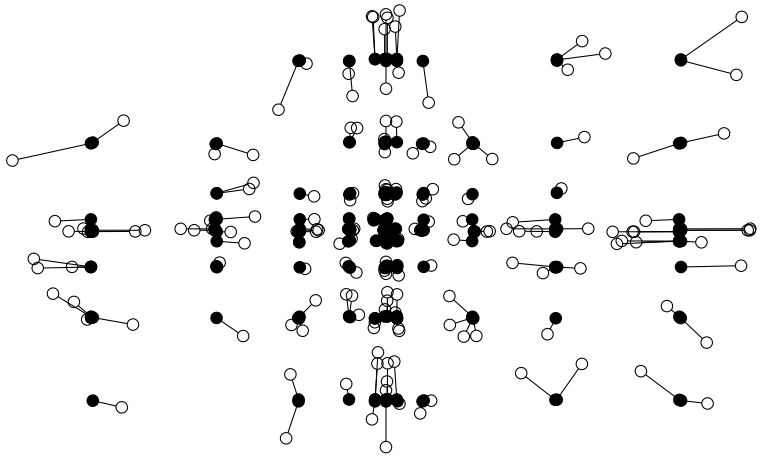

Figure 3.8: The bias after applying n to u.

3.4.5 Truncating the result

One possible approach to keep the computed results in a range where we are able to bound the computational errors thus avoiding the problem mentioned in the previous section, consists in truncating the result.

Some differential privacy mechanisms already use a truncation procedure.This traditional truncation works as follows: choose a subset $\mathbb{M}_r \subset \mathbb{R}^m$ and, whenever the reported answer x is outside $\mathbb{M}_r$ return the closest point to x in $\mathbb{M}_r$. However, while such a procedure is safe in the exact semantics because remapping does not alter differential privacy, the fact n and n' are not close prevent to rely on the computed result on which we do not control any property. So we cannot use this kind of truncation. Redrawing a new random value is not possible either because it would change the final distribution. So, to remain in a general framework where we do not have any additional knowledge about computational errors for large numbers, we decide here to return an exception value when the computed result is outside of some bounded subset $\mathbb{M}_r$ of $\mathbb{R}^m$. We represent the truncation in figure 3.9. The red dashed area represents ${\mathbb{M}_r}^c$, the set that we decide to truncate. Like in the previous figures, the small white circles correspond to the generated noise in the exact semantics while the black circles are the finitely generated noise. By returning an exception when the result is outside $\mathbb{M}_r$, we just keep results that are slightly deviated.

Raising an error means we loose significant utility. Better mechanisms can be found but they are specific to some algorithms or to some dimensions. For instance, we will show later that in the uni-dimensional case ($q = 1$) it is possible to return an extremal value because extremal values are into two disjoint sets. However, this truncation procedure has the advantage to be safe

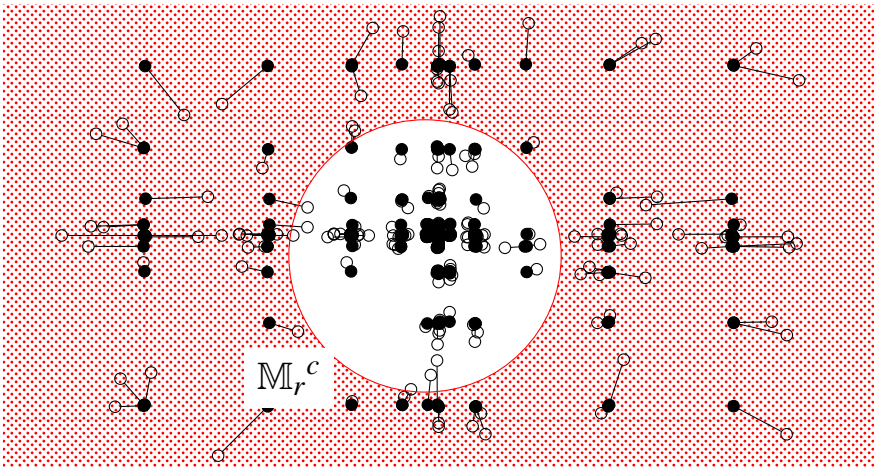

Figure 3.9: When the result is not in $\mathbb{M}_r$ (red area), we raise an exception.

in all cases.

We denote by ∞ the value returned by the mechanism when $f(D)+X \notin \mathbb{M}_r$. Hence, the truncated mechanism $\mathcal{A}$ returns the randomized value or ∞:

Mechanism 3.

$$\mathcal{A}_{\mathbb{M}_r}(D) = \begin{cases} f(D)+X & \text{if } f(D)+X \in \mathbb{M}_r \\ \infty & \text{otherwise} \end{cases}$$

We truncate the result because we want to exclude non-robust computations from our mechanism. However, to be efficient, this procedure need that computations from the input domain we define as unsafe provide output that are only in the unsafe domain where we raise an exception. To grant that erroneous outputs do not belong to the same domain as the safe ones, we need two more conditions. One requires the implementation to respect the monotonicity of the computed functions:

Condition 3.4.1. *This condition states that a function $g : \mathbb{R}^m \to \mathbb{R}^k$ and its implementation g' are such that for all $x, y \in \mathbb{R}^m$, $\|g(x)\| \leq \|g(y)\|$ implies $\|g'(x)\| \leq \|g'(y)\|$.*

With this property, if we know that the exact result corresponding to some obtained result is not in $\mathbb{M}_r$ then for any greater result we can obtain the exact result is also not in $\mathbb{M}_r$ either. If we do not require this property, we may have an implementation that returns 0 anytime an overflow occurs. Such an implementation would be valid because we do not ask for closeness outside of $\mathbb{M}_r$ but would generate a leakage since 0 would have a bigger probability to appear than expected.

The other condition is about the closeness of the implementation of the noise and its exact semantics in a safe area.

Condition 3.4.2. *The function n and its implementation n' are (k, δ_n)-close on some set U_r such that*

$$\forall u \in U_r^c \; f(D) + n(u) \notin \mathbb{M}_r{}^{+k\delta_0+\delta_n}$$

To find such a set U_r, one possible way is by a fix point construction. We begin by finding the smallest k_0 and δ_{n0} such that n and n' are (k_0, δ_{n0})-close on $\mathbb{M}_r$. Then for the generic step $m > 0$, we compute the smallest k_{m+1} and δ_{nm+1} such that n and n' are (k_{m+1}, δ_{nm+1})-close on $\mathbb{M}_r{}^{k_m\delta_0+\delta_{nm}}$.

If Conditions 3.4.1 and 3.4.2 hold, then from (3.1) we derive

$$\forall u \in U_r^c \; f'(D) + n'(n_0(u)) \notin \mathbb{M}_r \tag{3.4}$$

So whatever happens outside of U_r, the result will be truncated. We can then consider that there is no implementation error outside U_r. In other words, the implementation is observationally equivalent to a one such that n_0 is the identity on U_r^c and for all $u \in U_r^c$, $n(u) = n'(u)$.

Finally, we reformulate that fact into the following proposition.

Proposition 3.4.2. *When conditions 3.4.1 and 3.4.2 hold, the implementation of Mechanism 3 is equivalent to truncate the function $f(D) + n(u)$ whose implementation is such that*

$$\forall u \in U^q, d(f(D) + n(u), f(D) + n'(n_0(u))) \leq \delta_t$$

where $\delta_t = k\delta_0 + \delta_n$

3.4.6 Modeling the error : a distance between distributions

In the last section, we have bounded the error between the exact semantics and the finite-precision one. But this error was expressed in term of maximal error of computation i.e. we have bounded the distance between two functions. However, since we want to bound probabilities we have to get properties in terms of probabilities instead of maximal error.

So, we first define the measures associated to X and X'.

Definition 3.4.1. *We denote by μ and ν the probability measure of X and X', respectively: for all $S \in \mathcal{S}$ (where $\mathcal{S}$ is the set of measurable sets),*

$$\mu(S) = P[X \in S]$$

and

$$\nu(S) = P[X' \in S]$$

Given that we are in a probabilistic setting, the errors due to finite representation cannot be measured in terms of numerical difference as they can be in the deterministic case, they should rather be measured in terms of distance between the theoretical distribution and the actual distribution. Hence, we need a notion of distance between distributions. There exists several distance between probability laws, and in particular there is a family of distance called Wassertein distances. This is particularly interesting for our purpose because, as we will see, it has a direct relation with the computation error.

In the following, we use $\Gamma(\mu,\nu)$ to denote the collection of all measures on $\mathbb{R}^m \times \mathbb{R}^m$ with marginal μ and ν respectively.

Definition 3.4.2 (p-Wassertein distances)**.** *Let μ,ν two probability measures on $\mathbb{R}^m$ such that, for some $x_0 \in \mathbb{R}^m$,*

$$\int_{\mathbb{R}^m} d(x,x_O)^p \mathrm{d}\mu(x) < +\infty$$

, we have:

$$W_p(\mu,\nu) := \left(\inf_{\gamma\in\Gamma(\mu,\nu)} \int_{(\mathbb{R}^m)^2} d(x,y)^p \, \mathrm{d}\gamma(x,y) \right)^{1/p}$$

This definition is extended for $p = \infty$ as follows:

Definition 3.4.3 (∞-Wassertein distance [CDJ08])**.** *Let μ, ν two probability measures on $(\mathbb{R}^m, \mathcal{S})$ such that there exists a compact Ω, $\mu(\Omega) = \nu(\Omega) = 1$. the ∞-Wassertein distance between μ and ν is defined as follows:*

$$W_\infty(\mu,\nu) = \inf_{\gamma\in\Gamma(\mu,\nu)} \left(\inf_{t\geq 0} \left(\gamma\left(\left\{ (x,y) \in (\mathbb{R}^m)^2 \middle| d(x,y) > t \right\} \right) \right) = 0 \right)$$

We choose to use this ∞-Wassertein distance which, as we will show, is the natural metric to measure our deviation.

If we denote by $\mathrm{supp}(\gamma)$, the support where $\gamma(x,y)$ is non zero, we have an equivalent definition [CDJ08] for the ∞-Wassertein distance:

$$W_\infty(\mu,\nu) = \inf_{\gamma\in\Gamma(\mu,\nu)} \left(\sup_{\mathrm{supp}(\gamma)} d(x,y) \right)$$

We extend this definition to any pair of measures that differ only on a compact ($\mathbb{M}_r$ in our case) by considering the subset of $\Gamma(\mu,\nu)$ containing only measure $\gamma(x,y)$ with $\gamma(x,y) = 0$ if $x \neq y$ and either $x \in {\mathbb{M}_r}^c$ or $y \in {\mathbb{M}_r}^c$.

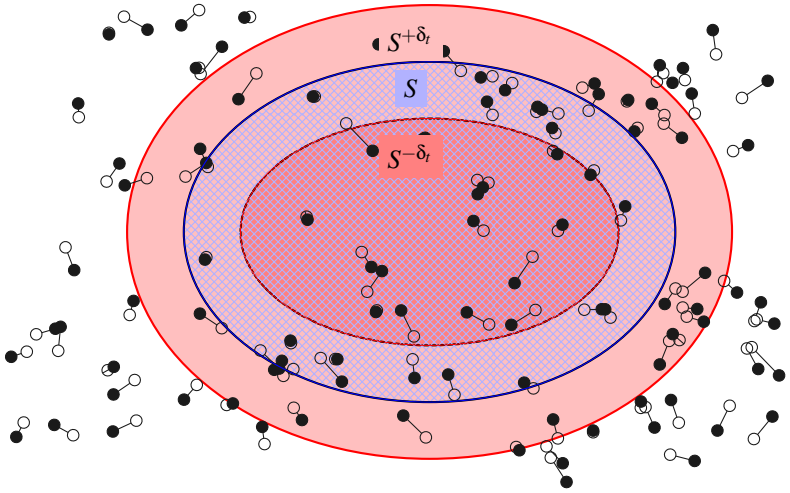

Figure 3.10: A bound on probabilities

We have introduced this measure because it has a direct relation with the computational error as expressed by the following proposition.

Proposition 3.4.3. *Let X and X' be two random variables $\Omega \to \mathbb{R}^m$ with distribution μ and ν respectively. We have that $\|X - X'\|_\infty \leq \delta$ implies $W_\infty(\mu,\nu) \leq \delta$.*

Proof. We consider the measure γ on $M \times M$, $\forall A,B \in \mathcal{S}, \gamma(A,B) = P(X \in A \wedge X' \in B)$. The marginals of γ are μ and ν. Moreover, the maximal distance between two elements in the support of γ is δ since $P(X \in A \wedge X' \in B) = 0$ when A and B are distant by more than δ. Since we have such a γ the minimum on all the $\gamma \in \Gamma(\mu,\nu)$ is less than δ. □

In our case, according to Proposition 3.4.2, we have $W_\infty(\mu,\nu) \leq \delta_t$. The following proposition allows us to bound the μ measure of some set with the measure ν.

Proposition 3.4.4. *Let μ,ν two probability measures on $\mathbb{R}^m$, we have:*

$$W_\infty(\mu,\nu) \leq \varepsilon \implies \forall S \in \mathcal{S}, \nu(S^{-\varepsilon}) \leq \mu(S) \leq \nu(S^{+\varepsilon})$$

Proof. The property of marginals is $\nu(S) = \int_{\mathbb{R}^m \times S} d\gamma(x,y)$. Since $\gamma(x,y) = 0$ if $d(x,y) > \varepsilon$, we derive $\nu(S) = \int_{S^\varepsilon \times S} d\gamma(x,y)$. Then we get $\nu(S) \leq \int_{S^\varepsilon \times \mathbb{R}^m} d\gamma(x,y)$. The last expression is the marginal of γ in S^ε, hence by definition of marginal: $\nu(S) \leq \mu(S^\varepsilon)$. The other inequality is obtain by considering $S^c = \mathbb{R}^m \setminus S$. □

In Figure 3.10, we illustrate the link between the computation error and the probability bound. The white circles represents possible results in the exact semantics and the linked black

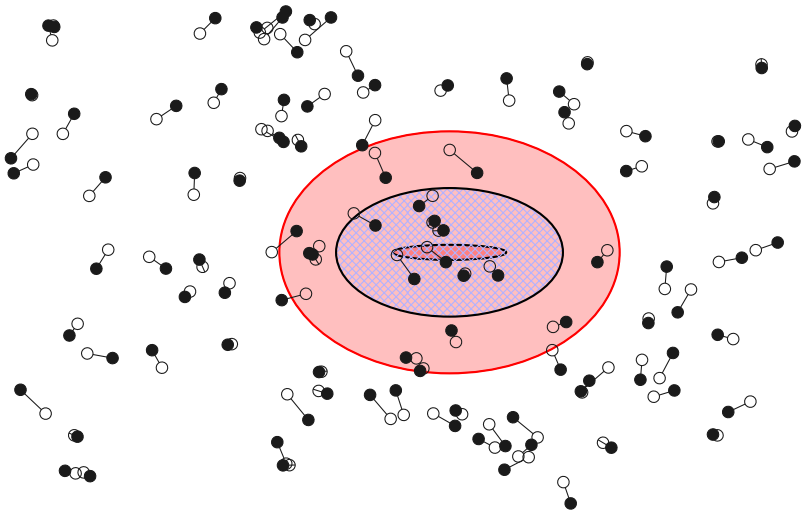

Figure 3.11: The problem of very small sets

circles represents the result computed in finite precision. All these links represent a mapping from $\mathbb{R}^m$ to $\mathbb{R}^m$. The probability that X' (in the actual semantics) belongs to some set S is equal to the probability that X (in the exact semantics) belongs to the set S^* which is the inverse image of S by this mapping. Since the norm of the links is bounded by δ_t, we get an over and an under approximation of S^* by expanding S or by shrinking S by δ_t (i.e. the sets $S^{+\delta_t}$ and $S^{-\delta_t}$).

3.4.7 Rounding the answer

The way we bound probabilities about X' by expanding and shrinking S gives suitable bounds when S is big enough. If S has a small area, $S^{-\delta_t}$ can be much smaller than S or even be the empty set as it is illustrated in Figure 3.11. It follows that the probability of the returned answer to be in $S^{-\delta_t}$ will be close to 0 or even null, and, since differential privacy is about ratio, this case will not be bound suitably.

We might argue that this problem is an artifact because we are modeling the computational errors through the ∞-Wassertein distance. This, however, is not the case. Indeed, this leakage is exactly the one described in example 3.1.1 where we actually built an attack.

For this reason, the analyst should be prevented to measure too small sets. The most natural way to achieve it consists in rounding the answer. Indeed rounding consists in mapping all values from a neighborhood to a unique value. Finally, it become impossible to measure sets that are smaller than these neighborhoods.

Therefore, once the computation of $\mathcal{A}(D)$ is achieved, we do not return the answer but a rounding of the answer. There will be no condition about this rounding, instead we will provide

a measure R of its strength that will be used to the measure of the overall leakage due to finite precision.

Mechanism 4. *The mechanism rounds the result by returning the value closest to $f(D)+n'$ in some discrete subset S'. So $\mathcal{K}(D) = r(\mathcal{A}(D))$ where r is the rounding function.*

From the above rounding function we define the set $\mathcal{S}'_0$ of all sets that have the same image under r. Then we define the σ-algebra $\mathcal{S}'$ generated by $\mathcal{S}'_0$: it is the closure under union of all these sets. Observe now that it is not possible for the user to measure the probability that the answer belongs to a set which is not in $\mathcal{S}'$. Hence our differential privacy property is now equivalent to the same formula where $\mathcal{S}$ has been replaced by $\mathcal{S}'$:

$$\forall S \in \mathcal{S}', P[\mathcal{A}(D_1) \in S] \leq e^{\varepsilon} P[\mathcal{A}(D_2) \in S] \tag{3.5}$$

In this way, we grant that any measurable set has a minimal measure and we prevent the inequality from being violated when the probabilities are small.

We still have to quantify this rounding to be able to measure the leakage in function of the rounding function used.

Definition 3.4.4 (Rounding ratio)**.** *We define R to be the maximal ratio between the areas $S^{+\delta_t}$ and $S^{-\delta_t}$ over all values that can be returned:*

$$R = \max_{S \in \mathcal{S}'_0, S \neq \emptyset} \frac{\lambda(S^{\delta_t} \setminus S^{-\delta_t})}{\lambda(S^{-\delta_t})} \tag{3.6}$$

This definition may look quite hard to compute, however, in case the rounding function is quite regular, we have the following proposition.

Proposition 3.4.5. *Let $(x_1, \ldots, x_m) \in \mathbb{R}$. If the chosen rounding function belongs to the following family*

$$\mathrm{rnd}_l(x_1, \vdots x_m) = \left(\frac{\lfloor l x_1 \rfloor}{l}, \ldots, \frac{\lfloor l x_m \rfloor}{l}\right)$$

then

$$R \leq m \frac{4\delta_t}{l + 2\delta_t} \left(\frac{l + 2\delta_t}{l - 2\delta_t}\right)^m$$

whatever is the used distance (definition 2.2.5).

Proof. With such a rounding, the space is split in hypercubes C of uniform size l. Therefore C^{δ_t} is include in a hypercube of size $l + 2\delta_t$ and $C^{-\delta_t}$ is a hypercube of size $l - 2\delta_t$. The Lebesgue measure of an hypercube of size s is by definition s^m. Then the measure of $C^{\delta_t} \setminus C^{-\delta_t}$ is $(l +$

$2\delta_t)^m - (l-2\delta_t)^m$. Then we have

$$\int_{l-2\delta_t}^{l+2\delta_t} mx^{m-1}dx = (l+2\delta_t)^m - (l-2\delta_t)^m$$

We can bound the integral by its higher value, we get:

$$(l+2\delta_t)^m - (l-2\delta_t)^m \leq 4\delta_t m(l+2\delta_t)^{m-1}$$

Finally, we conclude:

$$R \leq m\frac{4\delta_t}{l+2\delta_t}\left(\frac{l+2\delta_t}{l-2\delta_t}\right)^m$$

□

3.4.8 Strengthening the differential privacy property

The differential privacy expresses that the probability of reporting some answer differs by at most a factor e^{ε} when adding or removing one individual from the database. Once we decide to provide the differentially private mechanism 1 with a noise X that is absolutely continuous with the Lebegue measure ,i.e. there exists a distribution function p that describe X, this is equivalent to the following equation.

$$\forall x,y \in \mathbb{R}^m, d(x,y) \leq \Delta_f \implies p(x) \leq e^{\varepsilon}p(y) \tag{3.7}$$

But as we have seen in the subsection 3.4.6, when we want to bound some probability that X belongs to some set S, we need to know the probability it belongs to S^{δ_t}. This is problematic in the case in which the random variable X has a distribution p such that (3.7) holds, but it is not valid anymore when $\Delta_f \leq d(x,y) \leq \Delta_f + \delta_t$. In following example we illustrate the problem.

Example 3.4.3. *Consider a database with a field which can store values between* 0 *and* 1. *Consider the query that asks for the sum of this field over all the rows. We also assume that this summation is done without any computational error (which is the case for instance in the fixed-point representation). The sensitivity of this query is* 1 *since adding or removing a row will change the sum by at most* 1.

We decide to implement the mechanism 1 with a noise X with the following density distribution:

$$p(x) = Ke^{-\varepsilon\lfloor x \rfloor}$$

where K is the normalizing constant and $\lfloor x \rfloor$ *the function that returns the integer part of x.*

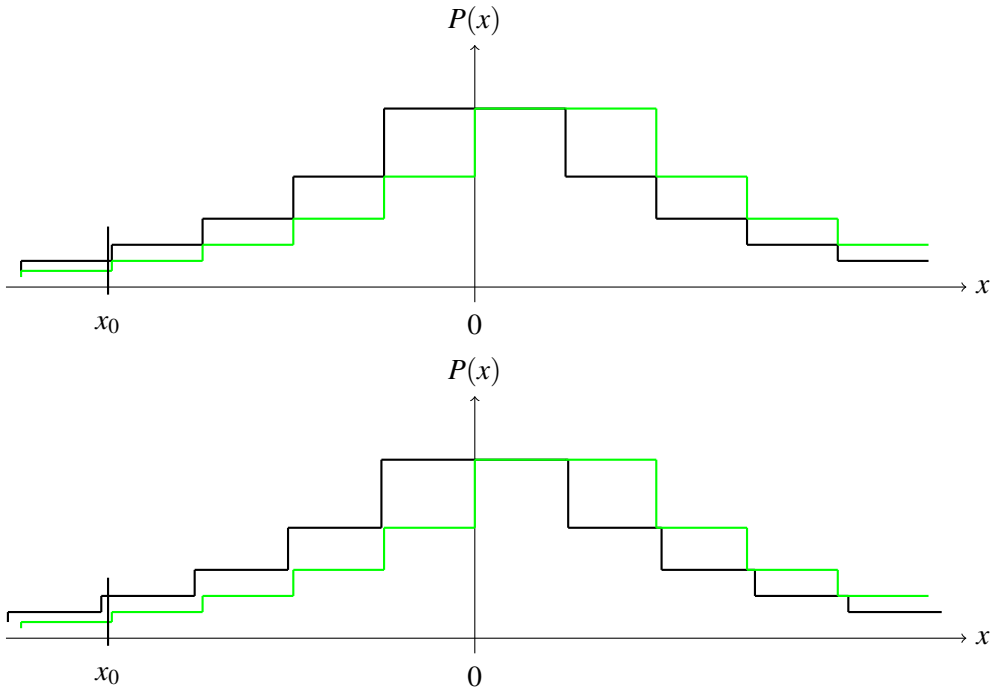

Figure 3.12: This distribution is differentially private in x_0 only in the exact semantics represented above.

This noise allows differential privacy since for all $x, y \in \mathbb{R}$ with $\|x-y\| \leq 1$:

$$p(x) \leq e^{\varepsilon} p(y)$$

However, a small computational error leads to a distribution slightly different as illustrated in figure 3.12. In this figure, we represent the distributions for two true answers 0 *and* 1 *either in the exact semantics (the graph above) or in the finite precision semantics (the graph below). In the finite precision, due to the deviation, there appear some small intervals where the ratio become twice higher than the maximum in the exact semantics (in x_0 for instance).*

In the last section, Proposition 3.4.4 uses the set $S^{+\delta_0}$ to bound the probability of the set S. In the previous example, when values differ by more than the sensitivity of the function, the e^{ε} ratio of differential privacy is not granted anymore leading to some leakage. In that case, we can only have a weaker ratio, by transitivity. Indeed, if $d(x,y) \leq 2\Delta_f$ then $p(x) \leq e^{2\varepsilon} p(y)$. So when the probability in $S^{+\delta_0} \setminus S$ is high, the differential privacy is broken (we loose a factor 2).

There are several ways to solve this leakage problem.

A first solution is to consider the following inequality as a new requirement replacing the

differential privacy property.

$$\forall \delta_1, \delta_2, |\delta_1| \leq \delta_t \wedge |\delta_2| \leq \delta_t, P[\mathcal{A}(D_1) + \delta_1 \in S] \leq e^{\varepsilon} P[\mathcal{A}(D_2) + \delta_2 \in S]$$

In case of a Laplacian noise, however, such a requirement will increase the differential factor ε by $\frac{\Delta_f + 2\delta_t}{\Delta_f}$ from the one obtained in the exact semantics. Indeed such a definition, is equivalent to assert that the sensitivity of the query is multiplied by $\frac{\Delta_f + 2\delta_t}{\Delta_f}$.

Another solution consists in using the fact that results are rounded. Indeed, in the previous example, we illustrate the fact that the error can be big (twice leakage than expected) but on a very small area. We might prove that this area A is bounded and cannot represent more than some fraction of the set S such that the influence of the factor 2 is limited by the small area where it applies. Then, the sketch of the proof is the following. First, we prove that $P(\mathcal{A}(D_2) \in A) \leq qP(\mathcal{A}(D_2) \in S)$. Then, we consider divide S in its inner part and its borderline part A:

$$P(\mathcal{A}(D_1) \in S) = P(\mathcal{A}(D_1) \in A) + P(\mathcal{A}(D_1) \in A)$$

Then we bound each of them.

$$P(\mathcal{A}(D_1) \in A) \leq 2e^{\varepsilon} P(\mathcal{A}(D_2) \in A)$$

$$P(\mathcal{A}(D_1) \in S \setminus A) \leq e^{\varepsilon} P(\mathcal{A}(D_2) \in S \setminus A)$$

Then we can conclude

$$P(\mathcal{A}(D_1) \in S) \leq (1+q) e^{\varepsilon} P(\mathcal{A}(D_2) \in S)$$

This solution is correct, but it adds a factor $(1+q)$. We chose, therefore, a third solution which constrains the distribution of X a little bit but which is optimal for the Laplacian noise because it does not add any multiplicative factor.

To achieve our purpose we need to impose some further condition on the mechanism.

Condition 3.4.3. *Given a mechanism $\mathcal{A}(D) = f(D) + X$, we say that $\mathcal{A}$ satisfies condition 3.4.3 with parameter ε (the desired parameter of differential privacy) if the random variable X has a probability distribution which is absolutely continuous according to the Lebesgue measure, and*

$$\forall S \in \mathcal{S}, r_1, r_2 \in \mathbb{R}^m, P[r_1 + X \in S] \leq e^{\varepsilon \frac{d(r_1, r_2)}{\Delta_f}} P[r_2 + X \in S]$$

This property is actually stronger than differential privacy as we state in the following proposition.

Proposition 3.4.6. *Condition 3.4.3 implies that the mechanism $\mathcal{A}(D) = f(D) + X$ is ε-differentially private.*

Proof. Let D_1 and D_2 be two databases such that $D_1 \sim D_2$. Let $r_1 = f(D_1)$ and $r_2 = f(D_2)$ be two answers. By definition of sensitivity, $d(r_1, r_2) \leq \Delta_f$ so $e^{\varepsilon \frac{d(r_1,r_2)}{\Delta_f}} \leq e^{\varepsilon}$. Hence,

$$P[\mathcal{A}(D_1) \in S] \leq e^{\varepsilon} P[\mathcal{A}(D_2) \in S]$$

□

We chose this condition because in case the noise has a Laplacian distribution then the converse of Proposition 3.4.6 holds and therefore in this case Condition 3.4.3 and differential privacy are equivalent.

Proposition 3.4.7. *Let $\mathcal{A}(D) = f(D) + X$ be a mechanism, and assume that X is Laplacian. If $\mathcal{A}$ is ε-differentially private (w.r.t. f), then Condition 3.4.3 holds.*

Proof. First, we show that if $\mathcal{A}$ is ε-differentially private then $b \leq \frac{\varepsilon}{\Delta_f}$ holds for the scale parameter b of X. Let $D_1 \sim D_2$ with $d(f(D_1), f(D_2)) = \Delta_f$. By ε-differential privacy we have, for any $S \in \mathcal{S}$:

$$P[f(D_1) + X \in S] \leq e^{\varepsilon} P[f(D_2) + X \in S]$$

From the density function of the Laplace noise (Definition 3.2.10), we derive:

$$K(n,d)d\lambda \leq e^{\varepsilon} K(n,d) e^{-b\Delta_f} d\lambda$$

Hence,

$$b \leq \frac{\varepsilon}{\Delta_f}. \tag{3.8}$$

Now, by definition of the density function, we have

$$P[r_2 + X \in S] = \int_{x \in S} K(n,d) e^{-bd(x,r_2)} d\lambda$$

From the triangular inequality, we derive:

$$P[r_2 + X \in S] \geq \int_{x \in S} K(n,d) e^{-b(d(x,r_1) + d(r_1,r_2))} d\lambda$$

Hence,

$$P[r_2 + X \in S] \geq e^{-bd(r_2,r_1)} \int_{x \in S} e^{-bd(r_1,x)} d\lambda$$

From inequality (3.8), we derive:

$$P[r_2 + X \in S] \geq e^{-\frac{\varepsilon d(r_2,r_1)}{\Delta_f}} \int_{x \in S} e^{-bd(r_1,x)} d\lambda$$

Finally,

$$P[r_2 + X \in S] \geq e^{-\frac{\varepsilon d(r_2,r_1)}{\Delta_f}} P[r_1 + X \in S]$$

□

3.5 Preserving differential privacy

Now, we have introduced a more robust mechanism and we have defined all the parameters we need to measure the leakage, we prove that if all conditions are met, then the finite-precision implementation of the mechanism satisfies differential privacy.

Theorem 3.5.1. *The finite-precision implementation of Mechanism 4 with a noise X that satisfies condition 3.4.3, a truncation procedure that satisfies condition 3.4.2 and a rounding function with parameter R, is ε'-differentially private, namely:*

$$\forall S \in \mathcal{S}, P[\mathcal{A}'(D_1) \in S] \leq e^{\varepsilon'} P[\mathcal{A}'(D_2) \in S]$$

where

$$\varepsilon' = \varepsilon + \ln(1 + Re^{\varepsilon \frac{L+\delta_t}{\Delta_{f'}}})$$

with $\delta_t = k\delta_0 + \delta_n$ and $L = \max_{S \in \mathcal{S}'_0} \varnothing S$.

Proof. Let S in $\mathcal{S}$. We first consider the case $S \neq \infty$.

Define $P_1 = P[\mathcal{A}'(D_1) \in S]$ and $P_2 = P[\mathcal{A}'(D_2) \in S]$. Since the result has been rounded (Definition 4), it is equivalent to consider the set $S' \in \mathcal{S}'$ with $S' = r^{-1}(S)$ instead of S.

Now we have $P_i = P[f'(D_i) + n'(X) \in S'] = P[n'(X) \in S' - f'(D_i)]$ where i is 1 or 2. Since ν is the measure associated to n', we have

$$P_i = \nu(S' - f'(D_i)).$$

From (3.4.2) and Theorem 3.4.3, $d(\nu, \mu) \leq \delta_t$. From Theorem 3.4.4 we derive

$$P_1 \leq \mu(S^{\delta_t} - f'(D_1)) \qquad \text{and} \qquad P_2 \geq \mu(S^{-\delta_t} - f'(D_2)).$$

The additivity property of measures grants us $\mu(S^{\delta_t}) = \mu(S^{-\delta_t}) + \mu(S^{\delta_t} - S^{-\delta_t})$. Condition 3.4.3

can be expressed in term of the measure as:

$$\forall S \in \mathcal{S}, r \in \mathbb{R}^m \|r\|, \mu(S) \leq e^{\varepsilon \frac{\|r\|}{\Delta_{f'}}} \mu(S-r).$$

From this inequality, we can derive, since $\|r\| = \Delta_{f'}$:

$$\mu(S^{\varepsilon}) \leq e^{\varepsilon} P_2 + \mu(S^{\delta_t} \setminus S^{-\delta_t}).$$

Since the probability is absolutely continuous according to the Lebesgue measure (Condition 3.4.3), we can express the probability with a density function p:

$$\forall S \in \mathcal{S}, \mu(S) = \int_S p(x) d\lambda$$

We derive:

$$\forall S \in \mathcal{S}, \min_{x \in S} p(x) \leq \frac{\mu(S)}{\lambda(S)}$$

By applying this property on $S^{-\delta_t} - f'(D_2)$, we get:

$$\min_{x \in S^{-\delta_t} - f'(D_2)} p(x) \leq \frac{\mu(S^{-\delta_t} - f'(D_2))}{\lambda(S^{-\delta_t} - f'(D_2))}$$

We derive:

$$\exists x_0 \in S - f'(D_2), p(x_0) \leq \frac{P_2}{\lambda(S)}$$

By the triangular inequality, we can bound the distance between x_0 and any point of S^{δ_t} by $\Delta_{f'} + L + \delta_t$ Hence, from Condition 3.4.3 we derive:

$$\forall x \in S^{\delta_t} - f'(D_1), p(x) \leq e^{\varepsilon \frac{\Delta_{f'} + L + \delta_t}{\Delta_{f'}}} p(x_0)$$

Then by integration:

$$\mu(S^{\delta_t} - f'(D_1) \setminus S^{-\delta_t}) \leq e^{\varepsilon \frac{\Delta_{f'} + L + \delta_t}{\Delta_{f'}}} \frac{\lambda(S^{\delta_t} \setminus S^{-\delta_t})}{\lambda(S^{-\delta_t})} P_2.$$

We rewrite this inequality according to definition 3.4.4:

$$\mu(S^{\delta_t} - f'(D_1) \setminus S^{-\delta_t}) \leq e^{\varepsilon \frac{\Delta_{f'} + L + \delta_t}{\Delta_{f'}}} R P_2.$$

Finally we obtain :

$$P_1 \leq (1 + Re^{\varepsilon \frac{L+\delta_t}{\Delta_{f'}}})e^{\varepsilon} P_2.$$

In case S is ∞, due to (3.4), $P[\mathcal{A}'(D) = \infty]$ is the same as $P[f'(D) + X' \in \mathbb{M}_r{}^c]$ where $d(\mu, \nu) \leq \delta_t$. Moreover, $\mathbb{M}_r{}^c$ can be decomposed in a enumerable disjoint union of element of $\mathcal{S}_0$. Therefore, the first part of the proof applies: ε'-differential privacy holds for all these elements. By the additivity of the measure of disjoint unions, we conclude.

□

3.6 Application of the Laplacian noise in one dimension

In our main theorem 3.5.1, we state a very general result which is parametric in the dimension m of the range space, in the law X of the added noise and on the implementation of this noise. Now, we consider the original case for which differential privacy has been used for and for which is the mostly used mechanism i.e. the case where $m = 1$ and X is distributed according to the standard linear Laplacian. This specific case will allow us to quantify the loss numerically. We show that the proposed mechanism 4 is still unsafe. Indeed, as we explain further, a set $\mathbb{M}_r$ that would prevent a large error δ_t would be too small to be useful. To solve this problem, we propose an improvement on the implementation of the random noise and show that, in this new setting, the loss is negligible.

3.6.1 Requirements and architecture assumptions

To be able to give numerical result we need to define which mechanism we use and where it is implemented.

Differential privacy mechanism We consider some intended degree of privacy ε. We consider the function f ranges over some interval $\mathbb{M}_r = [m, M]$. We denote by $r = M - m$ the size of this interval. We also express the sensitivity in function of a parameter N: $\Delta_f = r/N$. If the query is the summation of some subset of the entries, N represents the minimal number of entries that will be in the subset by the query.

A Laplace noise that provides ε-differential privacy in the exact semantics must have scale parameter Δ_f/ε. Now, to respect the pattern of mechanism 4, we need to decide the truncation range and the rounding function. Since f ranges in $[m, M]$, we truncate the result and return an error if we output a result outside of this range. For this example, we decide to round the result so that the result is a multiple of $\frac{r}{2^{52-s}}$ where s is an integer representing the number of significant digits that are removed.

Assumptions about the machine architecture We consider that the mechanism is implemented into a machine with the floating-point architecture following the IEEE standard [IEE08] on 64 bits.

We assume that the uniform generator used here is the one of the Java random library 3.4 i.e. it returns a multiple of 2^{-53}.

The next step consists on picking a number according to the Laplace distribution. The easier way to achieve this is to use the following formula.

$$X = n(U) = -b\,\mathrm{sgn}(U-1/2)\ln(1-2|U-1/2|)$$

Indeed, if U is a random uniform variable then X is a centered Laplacian distribution with scale parameter b. This standard technique to get a Laplace distribution is very convenient since its implementation just uses arithmetic operators, absolute values and the logarithm function that can be implemented without any loss of precision [HD93].

$$n(u) = \frac{\Delta_f}{\varepsilon}\mathrm{sgn}(u-1/2)\ln(1-2|u-1/2|). \tag{3.9}$$

The computation of the logarithm is in two steps. First, from the representation of $u = u_m 2^{u_e}$ where u_m is the mantissa and u_e the exponent of u, we have $\ln(u) = u_e \ln(2) + \ln(u_m)$. Hence, we reduce the problem to computing the logarithm for numbers in $[1,2[$. This computation is achieved by the CORDIC algorithm described in the forward section 4.1.1. This algorithm has been proved full-precision for some implementations [KC93], meaning that the error made by the algorithm is just the one due to the finite representation.

3.6.2 Weakness of the mechanism

Initial uniform noise As we explained in the subsection 3.4.3, since the uniform random generator returns numbers that are multiple of 2^{-53}, the δ_0 parameter defined in (3.1) is $\delta_0 = 2^{-53}$.

Closeness of n and n' In order to apply our theorem, we need to prove that condition 3.4.2 is satisfied. By proposition 2.3.5, it is sufficient to prove that, in the interval of interest, $n(u)$ is k-Lipschitz and that $|n(u) - n'(u)| \leq \delta_n$.

Proposition 3.6.1. *In our case, n is k-Lipschitz with*

$$k = \frac{2\Delta_f e^{\frac{\varepsilon r}{\Delta_f}}}{\varepsilon}$$

Proof. Since the result $n(u)$ is always truncated if the final result of our mechanism is outside of $[m,M]$, this means, since $f(D)$ belongs to $[m,M]$, that the result of $n(u)$ is always truncated when $|n(u)| \geq r = (M-m)$. So we are interested to know the k factor for which n is k-Lipschitz in $n^{-1}([-2r,2r])$. Since $n \in C^1$ (its derivative is continuous), it is enough to compute the maximal value of its derivative in this interval. So first we have to compute $n^{-1}([-2r,2r])$. $|n(u)| = r$ is equivalent to:

$$\left|\frac{\Delta_f}{\varepsilon}\operatorname{sgn}\left(u-\frac{1}{2}\right)\ln\left(1-2\left|u-\frac{1}{2}\right|\right)\right| = r$$

We derive:

$$u = \frac{1}{2} \pm \left(\frac{1}{2} - \frac{1}{2}e^{-\frac{\varepsilon r}{\Delta_f}}\right) \tag{3.10}$$

We have:

$$\frac{\dot{\mathrm{n}}}{\dot{\mathrm{u}}} = \frac{\Delta_f}{2\varepsilon(1-2|u-1/2|)}$$

Finally, the derivative is maximal on the limit of the interval computed in (3.10):

$$\frac{\dot{\mathrm{n}}}{\dot{\mathrm{u}}} \leq \frac{2\Delta_f e^{\frac{\varepsilon r}{\Delta_f}}}{\varepsilon} = k$$

So our function n is k-Lipschitz for the previously defined value of k.

□

This factor is too big as we now illustrate in a numerical application. Indeed, as we can see, this factor depends exponentially on $\frac{\varepsilon r}{\Delta_f}$. Now, in case of a sum query where values are in the interval $[0,\Delta_f]$, if the database contains N rows the result will be in $[0,N\Delta_f]$. Since we would like the truncation not to truncate any possible true results, we would like $r = N\Delta_f$. Finally, we get $k = 2\frac{\Delta_f}{\varepsilon}e^{\varepsilon N}$.

We have now to compute $\delta_t = k\delta_0 + \delta_n$. We have, in our architecture $\delta_0 = 2^{-53}$. Since summations and multiplication of values only generate error up to their precision and that the log function is full precision, the computational error δ_n will be less than $2^{-53}\Delta_f/\varepsilon r$. Therefore it can be negligible.

Finally, we have $\delta_t \approx 2\frac{\Delta_f\delta_0}{\varepsilon}e^{\varepsilon N}$.

Rounding the result The parameter 3.4.4 definition is

$$R = \frac{4\delta_t}{L-2\delta_t}.$$

This implies that, at least, $2\delta_t < L$. On the other hand, L should be not bigger than Δ_f/ε

otherwise the impact of the rounding on the utility of the answer would be bigger than the one due to the noise. This means that we have $2\delta_t < {}^{\Delta_f}/\varepsilon$. We expand this inequality:

$$4\frac{\Delta_f \delta_0}{\varepsilon} e^{\varepsilon N} < \frac{\Delta_f}{\varepsilon}$$

We derive:

$$\varepsilon N < 51\ln(2)$$

The standard deviation of the exact mechanism is $d = {}^{\Delta_f}/\varepsilon$ and corresponds to the average shift between the true value and the returned value, hence, the ratio $D = {}^{d}/_{\Delta_f}$ corresponds to some "relative expected deviation". From the last inequality and the fact that $51\ln(2) < 36$, we get: $D > {}^{N}/_{36}$. Since to be useful the relative deviation should be lesser than 1, it means the protocol does not work for query summing more than 36 rows, which is not acceptable (N is normally greater than one hundred).

3.6.3 Improvement of the implementation

The last mechanism was not able to provide accurate results in a large range. To solve this problem, we need the uniform generator to be able to return numbers smaller than 2^{-53}. One way to do it consist in generating the mantissa of the number in the classical way and then generating the exponent according to an exponential law as in [Mir12]. One slightly different way to proceed consists in using a formula equivalent to (3.9):

$$n(u,v) = \frac{\Delta_f}{\varepsilon} v \ln(|u|). \tag{3.11}$$

where $v \in \{-1,1\}$ and $P(v=1) = 0.5$.

If we decompose u between its mantissa $1+u_m$ and its exponent $-(1+u_e)$ which is a positive integer, we obtain:

$$n(u,v) = \frac{\Delta_f}{\varepsilon} v \ln(|(1+u_m)2^{-(1+u_e)}|).$$

Therefore, we derive:

$$n(u_m,u_e,v) = \frac{\Delta_f}{\varepsilon} v \left(\ln\left(|1+u_m|\right) - \ln(2)(1+u_e)\right)$$

If we want U to be uniform, then the probability to pick $u = (1+u_m)2^{-(1+u_e)}$ has to be proportional to the interval in the exact semantics which is rounded in u for that u_m has to be uniform and u_e has to be picked according to an exponential law i.e. $P(u_e = n+1) = 2^{-(1+u_e)}$.

There is a simple way to generate such a random variable u_e:

```
count = 0;
while( random_bit )
  count++ ;
return count;
```

where `random_bit` can be 0 or 1 with probability $1/2$.

Since there is no rounding error when computing integer random values, we only have to consider errors depending on u_m. But, now, for $u_m \in [0,1]$, $n(u_m)$ is $\frac{\Delta_f}{\varepsilon}$-Lipschitz.

Finally, the total error δ_t will be of the same order than the precision of the returned result i.e $\delta_t = 2^{-52}r$.

The parameter R defined in 3.4.4 is now very close to 0.

$$R = \frac{4\delta_t}{L - 2\delta_t}$$

Since $L = 2^{-52+s}r$ and $\delta_t = 2^{-52}r$, we derive:

$$R = \frac{4}{2^s - 2}$$

Finally the ε' parameter

$$\varepsilon' = \varepsilon + \ln(1 + Re^{\varepsilon\frac{L+\delta_t}{\Delta_f}})$$

can be rewritten with the computed values:

$$\varepsilon' = \varepsilon + \ln(1 + \frac{4}{2^s - 2}e^{\varepsilon\frac{L+\delta_t}{\Delta_f}})$$

If we set $s = 22$, then we have $2^s \gg 1$. By approximating $1 + 2^s \approx 2^s$ we get:

$$\varepsilon' \approx \varepsilon + \ln(1 + 2^{-s+2}e^{\varepsilon\frac{2^s r}{2^{52}\Delta_f}})$$

When we use the parameter $N = r/\Delta_f$:

$$\varepsilon' \approx \varepsilon + \ln(1 + 2^{-s+2}e^{\varepsilon\frac{2^s N}{2^{52}}})$$

Since N is an indicator of the number of the entries in the database, we can assume that

$$\varepsilon N \ll 2^{30}$$

So the exponential is approximately equals to 1.

Then since $\ln(1+x) \leq x$, and $2^{10} \approx 10^3$ we conclude that

$$\varepsilon' \approx \varepsilon + 10^{-6}$$

□

3.7 Application to the Laplacian noise in two dimensions

One of the nice features of our theorem 3.5.1 is its ability to deal with multi-dimensional variables. Here, we present the case in which queries return points in an Euclidean plan. This domain is used, for instance, in geo-location (see [ABCP13]). We consider the same architecture as in the previous section. The new difficulty, here, is to generate a random variable with a bivariate Laplace distribution. Since our definition of the multivariate Laplacian is not a standard one, there is no generation protocol in the literature. To generate this noise, we can generate it in polar coordinates and then converts it into Cartesian ones. Another option consists in generating the bivariate Laplacian for the d_1 distance. In the rest of this chapter, we will follow the first option, but we want to discuss briefly also this second option, for the sake of completeness.

This second option relies on the fact that

$$d_2(x,y) \leq \sqrt{2} d_1(x,y).$$

So if we generate a noise which is $\sqrt{2}\varepsilon$-private for the distance d_1, then it is also ε-private for the Euclidean distance d_2. This noise is easy to implement since two independent Laplace distributed random variables (X,Y) follows this bivariate Laplacian law. Indeed, we have

$$p(x,y) = Ke^{-bx}e^{-by} = Ce^{-(x+y)}$$

where K is the normalizing constant. So if we accept to loose a $\sqrt{2}$ factor either in the degree of privacy or in the utility we can just use the same method as in the previous section. We will not develop further this implementation since the purpose of this section consists in presenting difficulties that can happen with the proposal in [ABCP13].

We now focus on the first option. The probability density function in Cartesian coordinates is

$$p(x,y) = Ke^{b\sqrt{|x-x_0|^2+|y-y_0|^2}}$$

Following [ABCP13], we consider a transformation to polar coordinates. The radial probability

density function is expressed by the following formula:

$$p(r,\theta) = \frac{b^2}{2\pi} r e^{br}$$

Hence the cumulative function for the radius is:

$$C_b(r) = 1 - (1+br)e^{-br}$$

To generate the random variable, we have to compute $r = C_b^{-1}(u)$. We have $C_b(r) = u$ is equivalent to:

$$-(1+br)e^{-(1+br)} = \frac{u-1}{b}$$

To solve this equation, we need to introduce the Lambert function defined as the reciprocal of the function

$$f(x) = xe^x$$

Since f is not injective, there actually exist two functions W_0 and W_{-1} such that

$$W(x)e^{W(x)} = x$$

Here we use the W_{-1} negative function defined on $[-1/e, 0[$. Finally, we get the equation:

$$r = -\frac{W_{-1}\left(\frac{u-1}{e}\right)+1}{b}$$

While Lambert functions are not algebraic, there exist iterative algorithms to compute them with arbitrary precision [CGH$^+$96]. The problem we had with the logarithm in the last section is still relevant here. Indeed, even if we can bound the computational errors, the maximal value that the initial generator can provide is not close enough to 1 to return very large values. In order to avoid a too narrow truncation domain, we might add an additional protocol. Here, however, we will neither describe such a protocol, nor actually compute the maximal error that can be expected with such an algorithm. In the following, we just assume that the error is at most some δ_r value.

Once we have got the radius, the angle is obtained by multiplying a uniform random variable in $[0,1[$ by 2π. Finally, we convert (r,θ) into Cartesian coordinates.

In conclusion, the noise function is

$$n(u_r,u_\theta) = \left(-\frac{W_{-1}\left(\frac{u_r-1}{e}\right)+1}{e}\cos(2\pi u_\theta), -\frac{W_{-1}\left(\frac{u_r-1}{e}\right)+1}{e}\sin(\theta)\right)$$

Truncation Since most of the time the domain studied is bound (for instance the public transportation of a city is inside the limit of the city), we can do a truncation. However, we recall that our truncation is made for robustness purpose and not just for utility reasons. Hence, if our domain of interest is a circle, we will not choose $\mathbb{M}_r$ to be the same circle because the probability that the truncation would return an exception would be too high (more than one half if the true result is on the circumference).

Robustness of n As in the previous section, we do not analyze an actual implementation but we care about the k factor used for Condition 3.4.2. First, we analyze for which $k_C(\varepsilon, \varnothing(\mathbb{M}_r))$ the function C_ε^{-1} is k-Lipschitz in $[0, \varnothing(\mathbb{M}_r)]$. Since C is differential, this question is equivalent to find the inverse of the minimal value taken by its derivative function on the interval $C_\varepsilon^{-1}([0, \varnothing(\mathbb{M}_r)])$. By computing this minimum value, we get:

$$K_C(\varepsilon, \varnothing(\mathbb{M}_r)) = \frac{e^{\varepsilon \varnothing(\mathbb{M}_r)}}{2\varepsilon + r\varepsilon^2}$$

On the other hand, the computation of θ is just a multiplication by 2π of the uniform generator hence $k_\theta = 2\pi$. Then, with the conversion $(r, \theta) \mapsto (r\cos(\theta), r\sin(\theta))$ from polar coordinates to Cartesian coordinates we obtain the global k factor:

$$k = \sqrt{K_C(\varepsilon, \varnothing(\mathbb{M}_r))^2 + 2\pi\varnothing(\mathbb{M}_r)}$$

Let δ_n be the distance between n and n', and δ_0 be the error of the uniform generator. From (3.4.2) we get:

$$\delta_t = \sqrt{K_C(\varepsilon, \varnothing(\mathbb{M}_r))^2 + 2\pi\varnothing(\mathbb{M}_r)}\delta_0 + \delta_n.$$

Rounding the answer We now compute the parameter R in (3.4.4). The rounding is made in the Cartesian coordinates, hence the inverse image of any returned value is a square S of length L. Note that S^{δ_t} is included in the square of length $L + 2\delta_t$ and $S^{-\delta_t}$ is a square of length $L - 2\delta_t$. Hence the ratio value is smaller than $R = (\frac{L+2\delta_t}{L-2\delta_t})^2$.

Differential privacy By Theorem 3.5.1 we get that (the implementation of) our mechanism is ε'-differentially private with

$$\varepsilon' = \varepsilon + \ln(1 + (\frac{L+2\delta_t}{L-2\delta_t})^2 e^{\varepsilon \frac{L+\delta_t}{\Delta_{f'}}})$$

3.8 Conclusion and future work

3.8.1 Conclusion

This chapter was concerned by differential privacy: a security protocol that relies on real-valued random numbers generation. Existing study on differential privacy focus either on its property on the exact semantics or on the technical problem raised by a particular implementation. In this chapter, we present a theoretical model for errors to study them at high level.

To do so, we model the error as a non deterministic bias. While in cryptographic protocols, adversaries have limited capacities (they cannot solve hard problem), here (except for dependency between random values), we make no assumptions: the analyst can have a perfect knowledge about how rounding errors occurs.

With this model, we have been able to prove that the additive mechanism 1 which is safe from a theoretical point of view breaks the differential privacy once implemented in finite precision architecture. This loss has two origins: first, from extreme values the errors are significant, and secondly, locally, last digits can provide fingerprints of the secret. Our solution solves these two problems: the last digits leakage has been fixed by a rounding procedure and the extreme perturbations by raising an error when the result is outside some values.

After proposing these fixes, we have done a quantitative analysis to measure the loss of privacy induced by finite-precision representation. We have proved that when the fixes on the mechanism are implemented the differential privacy parameter is just increased by some additive factor.

To analyze the relevance of our general result, we have applied it for the standard method (Laplacian in one dimension). This strict application was not really satisfactory: in general, the loss is too important. Therefore, we have proposed a last fix where the uniform random generator of mechanism 2 has been replaced by a dynamic procedure that provides as much random bits as necessary to grant differential privacy.

Finally, we have explained the steps for implementing bivariate Laplace noise. Here there is no obvious way to allow a random value to be generated in a large range.

3.8.2 Future work

As future developments of this work, we envisage two main lines of research:

- Deepening the study of the implementation error in differential privacy: there are several directions that seem interesting to pursue, including:
  - Improving the mechanisms for generating basic random variables. For instance,

when generating a one-dimensional random variable, it may have some advantage to pick more values from the uniform random generator, instead than just one (we recall that the standard method is to draw one uniformly distributed value in $]0,1]$ and then apply the inverse of the cumulative function). For instance, $u_1 + u_2$ has a density function with a triangular shape and costs only one addition. The other advantage is due to the finite representation: if the uniform random generator can pick N different values then two calls of it generate N^2 possibilities, which enlarge considerably the number of possibilities, and therefore reduce the "holes" in the distribution.

- Considering more relaxed versions of differential privacy, for instance the (ε, δ)-differential privacy allows for a (small) additive shift δ between the two likelihoods in Definition 3.3.3 and it is therefore more tolerant to the implementation error. It would be worth investigating for what values of δ (if any) the standard implementation of differential privacy is safe.

- Enlarging the scope of this study to the more general area of quantitative information flow. There are various notions of information leakage that have been considered in the computer security literature; the one considered in differential privacy is just one particular case. Without the pretense of being exhaustive, we mention the probabilistic approaches [PHW05, HO05, BP05], the information-theoretic approaches based on Shannon entropy [CHM05, Mal07, CPP08] and those based on Rényi min-entropy [Smi09, BCP09, BPP11, BP12] and the more recent approach based on decision theory [ACPS12]. The main difference between differential privacy and these other notions of leakage is that in the former any violation of the bound in the likelihood ratio is considered catastrophic, while the latter focuses on the average amount of leakage, and it is therefore less sensitive to the individual violations. However, even though the problem of the implementation error may be attenuated in general by the averaging, we expect that there are cases in which it may still represent a serious problem.

$\star$

CHAPTER 4

GLOBAL ANALYSIS OF PROGRAMS

Contents

In the previous chapter, we have seen how to prove a probabilistic property (namely differential privacy) on an implemented algorithm from our knowledge about its robustness.

In this chapter we are interested in how to prove that a given implementation of a program is robust. We consider both the $P(k,\varepsilon)$ property of the implemented semantics and the (k,ε)-closeness property between the exact and the finite precision semantics.

There already exist lot of methods to prove robustness of implementation. Some are based on formal logic like Hoare triple proofs. Others are based on abstract interpretation with zonotopes [Gou01]. Others are specialized in particular numerical programs like polynomial roots solver. These methods are either fully automatic or need human directives.

The work we present here investigates a new direction for analysis. Standard analysis either based on Hoare logic or abstract interpretation are mainly bottom-up analysis. Indeed, the global property for the program is provided by a composition of properties about single instructions. Here, we try a non compositional top-down approach. Going in the other direction means we already have knowledge about the context then we identify a pattern that allows to decompose some part of the code into thinner parts of codes. Providing such a general method is an ambitious project and we do not claim we have achieved it. In this chapter, we target a restrictive objective: how to apply such a given pattern on some code such that its decomposition allows to conclude on the robustness to the program. The main idea would be to be able to develop a library of patterns such that any kind of code can be decomposed in simpler parts until reaching basic code. Here, we focus on finding a pattern for one of the most difficult problem for compositional analyzers: programs that are robust only at the end of the execution but not in the middle. Indeed, systematic compositional analyzers try to preserve some knowledge about robustness all along the code. If, at some point, there is no robustness property that can hold then the robustness property is lost and cannot be retrieved afterward.

To study this problem and to investigate some solutions, we consider two examples of code with such a behavior: the CORDIC algorithm for computing trigonometric functions and the Dijkstra's shortest path graph algorithm. We choose these algorithms since they use general principles (actually, they are classes of algorithms since there are several variants of them), they are different from each other and they are relatively simple and short algorithms. We mainly propose two solutions and show how they apply to these two examples. The first solution is based only on the finite precision semantics. However, this solution was not so conclusive since the decomposition in smaller parts does not lead to a simpler analysis. So, we developed a second solution, which is based both on mathematical proofs on the exact semantics and on a analysis of the actual code in the finite precision semantics. Such hybrid analyze is justified because most algorithms are based on mathematical theorems that already exists. So, in general, this method should not require too much additional proof in the exact semantics to be completed.

This chapter is organized in the following manner. In section 4.1, we describe precisely the problem of programs that are not locally robust. We also present our two examples CORDIC and Dijkstra and we explain why they are challenging. In section 4.2, we do a general description on the way we define the patterns. In section 4.3, we present our first solution and in section 4.4, we present the second one based on the rewriting framework.

4.1 Presentation of the problem

The specific problem we investigate in this chapter is the problem of global behavior that are not local behavior. For instance, if a program only manipulates one variable, and all instructions make a continuous transformation on $x = f_i(x)$ (like `x= sin(x);`) then the whole program is also continuous because of the compositionality of continuity.

A program that is not locally continuous but globally continuous is a program which is not continuous at an intermediate stage. For instance, the code in figure 4.1 is only globally continuous. Indeed, if we stop its execution after the first branch the computed function is discontinuous in 0, while at the end, the computed function is $f(x) = x+2$ which is a continuous function.

Such programs cannot be analyzed line by line for any property that implies continuity like k-Lispchitz or $P(k,\varepsilon)$ properties. In addition, even if it may be quite easy to prove that the code interpreted in the exact semantics has the global behavior, it become much harder to prove the same property in the finite-precision semantics.

To analyze the problem in details, we propose two examples of programs that are actually used. These examples will be our tests for the solutions we propose in the next sections.

```
1   if(x < 1)
2     x = x - 1;
3   else
4     x = x + 1;
5   if(x < 0)
6     x = x + 3;
7   else
8     x = x + 1;
```

Figure 4.1: An example where the continuity property is lost line 4 but reappear line 8

4.1.1 Cordic

CORDIC (COordinate Rotation DIgital Computer) [Vol59] is a class of simple and efficient algorithms to compute hyperbolic and trigonometric functions using only basic arithmetic (addition, subtraction and shifts), plus table look-up. The notions behind this computing machinery were motivated by the need to calculate the trigonometric functions and their inverses in real time navigation systems. Still now-a-days, since the CORDIC algorithms requires only simple integer math, CORDIC is the preferred implementation of math functions on small hand calculators.

CORDIC is a successive approximation algorithm: a sequence of successively smaller rotations based on binary decisions drives the algorithm towards the value we want to find. The CORDIC version illustrated in the code 4.2 computes the cosine of any angle in $[0,\pi/2]$. In the code, the instruction << operates a shift of the bits to the left such that `x << i` computes $2^i x$. Note that this program makes call to trigonometric functions like cosine itself or arc tangent. In fact, whatever is the input, these functions are always called on the same inputs $\pm 2^{-i}$ for i in $[1,n]$. So, in practice, it is just a fixed set of constants that is stored by the program and not real instructions.

Before analyzing the consequences of computational errors in the CORDIC algorithm, we start by explaining its principle and prove its correctness in the exact semantics.

Presentation of the algorithm

To compute the cosine of a given angle β, the CORDIC algorithm computes the coordinates (x,y) of a point P on the unit circle by successive rotations until the polar coordinates of P are $(\alpha,1)$ with $\alpha \approx \beta$.

```
1  double cos(double beta) {
2      double x = 1, y = 0, theta = 0, x_new, sigma;
3      int n = 15, i;
4      for( i = 1 ; i < n ; i++ ) {
5           if( beta > theta )
6             sigma = 1;
7           else
8             sigma = -1;
9           sigma =  sigma / (1 << i);
10          theta = theta + atan(sigma); // Value stored
11          fact = cos(atan(sigma)); // Value stored
12          x_new = x + y * sigma;
13          y = fact * (y - x * sigma);
14          x = fact * x_new; }
15     return x; }
```

Figure 4.2: An implementation of CORDIC

During these iterations, the invariant is

$$(x, y) = (\cos\alpha, \sin\alpha)$$

Indeed, at each iteration, a predefined angle σ is chosen for which the trigonometric functions are inside a database. The point P is replaced by itself rotated by σ while α is incremented by α.

The choices of the σ angles are such that at the end of the process α is very close to β and then x and y are close to $\cos\beta$ and $\sin\beta$. This process is represented in figure 4.3. The successive positions of P and α are represented. In the exact semantics, α always corresponds to the angle of P with the x axis and P always belongs to the unit circle. Finally, the final value of α is close to β.

In this chapter, we consider two variations in the way σ values are chosen.

In the algorithm (code 4.2), the values of σ are $\pm\tan^{-1}2^i$ where i is the number of iterations. By doing this, the rotation can be expressed as (for a positive σ):

$$\begin{pmatrix} x \\ y \end{pmatrix} = \cos(\tan^{-1}(2^{-i})) \begin{pmatrix} 1 & 2^{-i} \\ -2^{-i} & 1 \end{pmatrix}$$

This calculus enjoys just addition and multiplication by a power of 2 (which is fast in a binary representation).

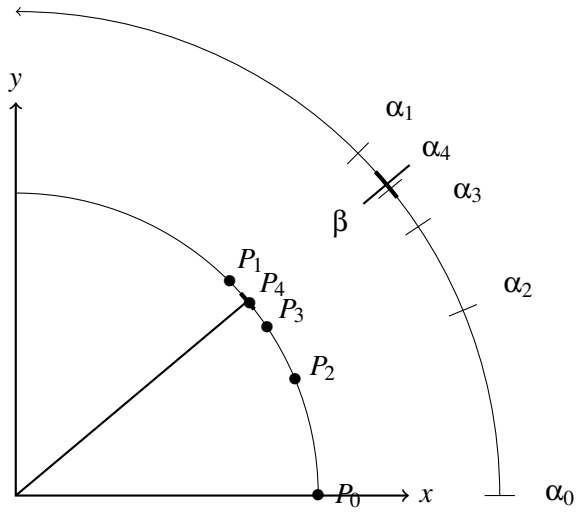

Figure 4.3: Iterative steps of the CORDIC algorithm in the exact semantics

In fact in the actual algorithm, the product of all factors $\cos(\tan^{-1}(2^{-i}))$ is stored since it does not depend on β. But, we do not present this version which is more complex to analyze.

Analysis of CORDIC algorithm in the exact semantics

The purpose of this chapter is not to study the exact semantics of CORDIC but what is happening when errors occur. However, if the algorithm itself is unsafe, there is no chance for the implementation to be correct. In addition, in section 4.4, we use the assumption that most implemented algorithms have already been proved in their mathematical setting and that we can use these proofs to study the behavior in the finite precision semantics. Here, we present the proof even if this algorithm has already been proved by several ways to stress the difference between proving a program in the exact semantics and proving it in the finite-precision semantics.

To prove the correction of the algorithm, we need two properties. First, we have to prove that the algorithm always terminates on its validity domain. Then, we have to prove that the final result corresponds to what is intended.

Proposition 4.1.1. *The CORDIC implementation terminates for all valid inputs.*

Proof. The number of iterations is constant equals to n. □

To prove the correctness of the algorithm, we have to provide a bound on the distance $d([\![\texttt{cos(beta)}]\!], \cos(\beta))$. First we need to find a bound on the distance between α the final value of `theta` and β.

Proposition 4.1.2 ([Vol59]). *At the end of the execution, we have:*

$$|\alpha - \beta| \leq \tan^{-1}(2^{n-1})$$

Proposition 4.1.3. *We can bound the distance between the result of the computation and the actual result.* $d([\![\texttt{cos (beta)}]\!], \cos(\beta))$

Proof. We already know that:

$$(x, y) = (\cos\alpha, \sin\alpha)$$

From proposition 4.1.2, $|\alpha - \beta| \leq \tan^{-1}(2^{n-1})$ and the fact cos is $\sqrt{2}$-Lipschitz function, we get:

$$d([\![\texttt{cos (beta)}]\!], \cos(\beta)) \leq \sqrt{2\tan^{-1}(2^{n-1})}$$

□

Proposition 4.1.4. *The Cordic implementation 4.2 is* $P(\sqrt{2}, 2^{-(n-1)}\sqrt{2})$ *in the exact semantics.*

Proof. From proposition 2.3.1, since cos is $\sqrt{2}$-Lipschitz and

$$d([\![\texttt{cos (beta)}]\!], \cos(\beta)) \leq \sqrt{2\tan^{-1}(2^{n-1})}$$

we get the result. □

The CORDIC algorithm in the finite-precision semantics

Now, we present how the behavior in finite precision can differ from the exact behavior.

First, the correctness of CORDIC relies on the invariant:

$$(x, y) = (\cos\alpha, \sin\alpha)$$

In finite precision, however, this invariant does not hold anymore since there are small variations at each iteration. In section 4.3 where we do a direct analysis of the finite precision, we have to consider that the invariant is replaced by

$$d((x, y), (\cos\alpha, \sin\alpha)) \leq \varepsilon$$

Since, we are looking for a quantitative analysis, this means we have to compute the value of ε at each iteration. However, this new definition complicates a lot the proof of the correctness of

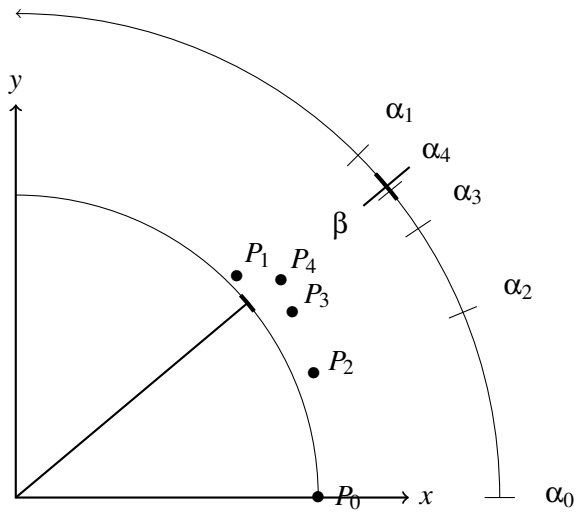

Figure 4.4: The invariant is broken

the algorithm. We illustrate in figure 4.4, this kind of deviation: now, the point P is just close to the unit circle.

The second problem is about the stopping condition. In our program, where the number of iterations is fixed, the termination is still granted. The final value that α should have, however, can become problematic. Indeed, when i become very small, an unsafe arithmetic might round the value such that the values of x, y and α do not change at all instead of changing just a little bit. If this phenomena happens, the reasoning we have done about the final value is not valid anymore. The phenomena is represented figure 4.5: the final value P_4 has now an angle not close to β.

In case the stopping condition is dependent on the value of α, for instance, with a condition like `while (| beta - theta | > e)` for some small constant e, the problem is even worse: we cannot grant the termination of the algorithm anymore.

The main concern of this chapter is not about these kinds of deviation, even if we have to deal with them, but about changes in the control flow. The change in the control flow is illustrated in figure 4.6. The problem appears when β is very close to $\pi/2$ for instance. In that case, while the exact semantics takes one branch of the `if`, the finite precision semantics takes the other one. This leads to two intermediate values that are completely different. However, such a behavior should not be considered as non robust since the final values can still be quite close to the true result.

Such a behavior is challenging for compositional analyzer: indeed, such analyzers tries to maintain a distance between the exact and the finite precision semantics at each step of the

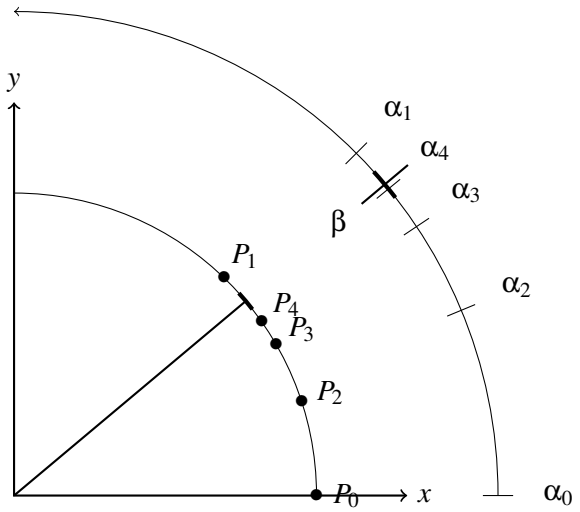

Figure 4.5: The termination fails

process. When they encounter a shift in the control flow, the distance cannot be maintain anymore so they return a failure. The main goal of this chapter is to propose an alternative when compositional analysis is structurally inefficient, like in this case.

4.1.2 Dijkstra's shortest path algorithm

Our second working example is the Dijkstra's shortest path algorithm [Dij71].

This graph algorithm allows to compute the minimal path from a given node to any node of the graph. In its traditional formulation, the length is defined as the number of edges to link the source and the destination.

Here, as we are interested in real valued algorithms, we consider the extension of the algorithm where a real valued length is associated to each edge such that the distance for a given path is the sum of these lengths.

To compute this minimal distance, the algorithm associates an estimated distance to each node. At the beginning, the source node is associated to 0 and all other nodes are set to ∞. The algorithm also stores the nodes that have already been processed. No node are marked as processed at the beginning.

The iterative step is the following (see figure 4.8). The algorithm looks for the node among the non-processed nodes, which one has the minimal estimated distance. From this node, it computes an estimated distance for all its neighbors: if this distance is better than the older one, the estimated value for the node is updated. Once, this computation is done for all the neighbors, the node is marked as processed.

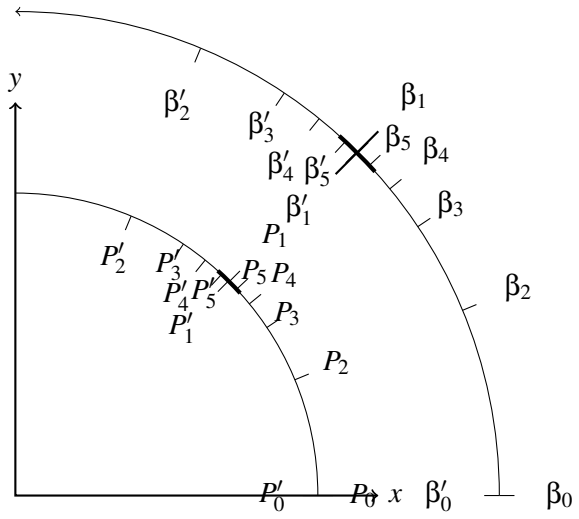

Figure 4.6: The termination fails

The algorithm stops when all node are marked (see figure 4.9). An implementation of the Dijkstra's algorithm in the C language is given in figure 4.7.

In this implementation, we use some conventions: the number of nodes is fixed to w and the maximum value for a path is 999 (some stand-in of infinity).

For this algorithm we do not recall the proof of its correction. It can be found in [Dij71]. We only stress that such an algorithm works only with positive values for the edges, otherwise it is not even possible to define what is the minimal path (due to the possibility of infinite circles).

Dijkstra's in the finite precision semantics

The Dijkstra's algorithm is more robust to errors than the CORDIC algorithm. Indeed, it does not rely on some analogical invariant: in finite precision even if values are shifted, the fact that marked nodes have smaller values than non marked one is still preserved.

However, such a program can also have a very different control flow for its finite precision semantics compared to the exact one. Indeed, when lengths change a little bit, the node with the minimal value can change. Then, if the node with minimal value changes, the next update is not done at the same nodes so that the intermediate values after i iterations can differ a lot between the two semantics. For instance, in one case, the estimated value for a node has not been done so the value is still 999 while in the exact semantics, the node has been processed and then has a smaller value. So, in this algorithm too, compositional analysis is not possible and another solution has to be found.

```
int[] dijkstra( int graph[w][w]){
   int pathestimate[w],mark[w];
   int source,i,j,u,predecessor[w],count=0;
   int minimum(int a[],int m[],int k);
   for(j=1;j<=w;j++){
      mark[j]=0;
      pathestimate[j]=999;
      predecessor[j]=0;}
   source=0;
   pathestimate[source]=0;
   while(count<w){
     u=minimum(pathestimate,mark,w);
     mark[u]=1;
     count=count+1;
     for(i=1;i<=w;i++){
        if(pathestimate[i]>pathestimate[u]+graph[u][i]){
            pathestimate[i]=pathestimate[u]+graph[u][i];
            predecessor[i]=u;}}}
     return pathestimate;}

int minimum(int a[],int m[],int k){
     int mi=999;
     int i,t;
     for(i=1;i<=k;i++){
          if(m[i]!=1){
               if(mi>=a[i]){
                    mi=a[i];
                    t=i;}}}
     return t;}
```

Figure 4.7: An implementation of the Dijkstra's algorithm

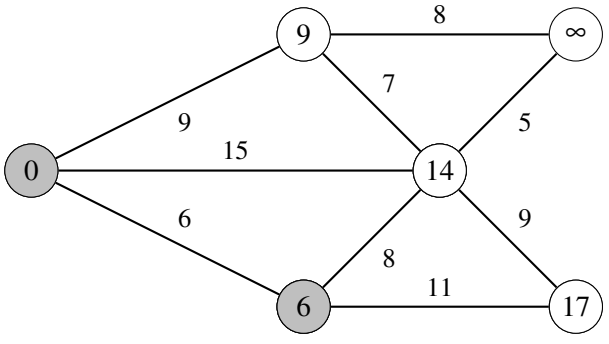

Figure 4.8: Dijkstra's algorithm after two steps

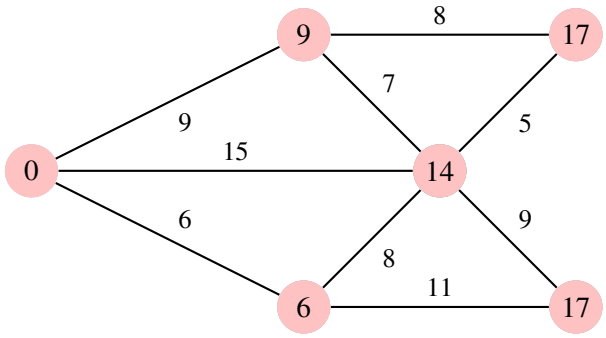

Figure 4.9: Dijkstra's algorithm final state

```
if(cond(x))                    for(i=0;i<n;i++)
  for(i=0;i<n;i++){              if(cond(x)){
  instr;                       instr;
}                              }
```

Figure 4.10: Equivalent programs that do not have the same syntactic structure

4.2 Semantic pattern matching

A solution, when a program is not locally robust, may be to rewrite it in a higher level language where the compositional analysis may apply. Such solutions have been proposed in [CGL10] and this method actually applies for the Dijkstra's algorithm (without being general enough to deal with the CORDIC algorithm). However, this solution is heavy since the program has to be written again. There is also another concern we decided to investigate. Indeed, there exist programs that are computationally equivalent while they differ only by the syntax. For instance, the two programs in figure 4.10 are equivalent while the `if` branch and the `for` loop are inverted in case the instructions in the loop do not change the value of `x`. If our analyzer only follows the syntax, it may fail in one case while it succeeds in the other case. Moreover, sometimes the analysis of a program would be easier with an additional syntactic constructor. Such a trick is not suitable, since maybe another analysis of the code (to prove another property than robustness) would be harder with this construct than with the original code. From these remarks, it appears that a pattern based only on the syntax is not sufficient. So, instead of the classic pattern based on the syntax, we would like to use pattern based on the trace semantics of the program.

However, we do not develop a systematic pattern matching mechanism in this work but we introduce some notions for our particular purpose.

In order to do the pattern matching, we need to use an abstract language. This language only contains information about control flow and communication between processes.

To define interacting processes from a single process, we first split the memory space in a partition, each part is represented by an abstract variable. Once such a partition is defined, it is possible to categorize instructions depending on which partition is read and which one is written. Hence, in our abstract language, one instruction can match a block of instructions of the concrete language such that all of them access and modify the same sets of variables:

```
(x,y) = O(a,b,c);
```

This abstract code denotes a collection of program instructions `O` that modifies values of the abstract variables **`x`** and **`y`** while accessing only the variables **`a`**, **`b`** and **`c`**. If a concrete variable has a local scope, it can be considered as a local variable within the collection of instructions `(x,y)=O(a,b,c)` and not part of the global variables. For instance, the instructions

```
{ int x = z * z;
y = z + x;
} // end of the scope of x
```

can match the abstract code

```
(y) = T(z);
```

even if it contains `x` because `x` does not have any access to other instructions than the ones in `T`.

Program inputs and outputs are specified in the following way.

```
foo(i,j){
  ...
  return r,s;}
```

These declaration lines mean that the concrete parameters of the function should belong to the union of the abstract variables **`i`** and **`j`** and that the set of the returned concrete variables are included in **`r`** and **`s`**.

For each abstract variable **`a`**, we use the following notations.

- We denote by a, the value of **`a`** at some point to be specified.
- We denote by $\mathcal{D}_a$, the domain to which belongs a. Given that the abstract variable **`a`** can correspond to n_a variables (n_a depends on the matching), and since these n_a variables are finite representations of real numbers, $\mathcal{D}_a$ is essentially $\mathbb{R}^{n_a}$ (the cross product $\mathbb{R}$ n_a times). In some cases, however, an abstract variable can instantiate integers. In this particular case, $\mathcal{D}_a$ is a cross product of $\mathbb{N}$.
- We denote by d_a a distance on $\mathcal{D}_a$. For the input and the output, this distance has to be consistent with one of the studied property ($P(k,\varepsilon)$ and (k,ε)-property implicitly depends

```
foo(i){
      a = a0;
      b = b0;
      while(S(i,a)){
            c = O(a,b,c,i);
            a = M(a,c);
            b = N(i,b,c);
      }
      return b; }
```

Figure 4.11: The main template

on a distance on the input and the output). For the other variables, the distance can be chosen freely. However, in most analysis here, we only use the d_1 distance (definition 2.2.5).

- We use the Kleene star notation $\mathcal{D}_a^*$ to denote the set of all finite lists over $\mathcal{D}_a$.

4.3 A direct analysis of the finite precision semantics through program transformation

In this first pattern, we do not consider the exact semantics but only the finite precision one. Here, we prove the definition 2.3.2 of robustness i.e. the $P(k,\varepsilon,\delta)$ property. Such an analysis is interesting in case we do not have any knowledge about the program and we try to discover a minimal set of properties that grants the robustness result.

4.3.1 The schema structure

The main characteristic of this first pattern is to produce, from an initial program containing an `if` branch, another program that does not contain the `if` branch. To achieve this "removal", the body of the `while` loop is split into three parts. Then, owing to some transformations, we reduce the analysis of the whole program to simpler programs without the instructions in `O(a,b,c,i);` that contain the `if` branch: once the new programs are proved $P(k,\varepsilon,\delta)$ with a compositional analysis, our theorem states that the whole program is $P(k',\varepsilon',\delta')$ with parameters depending on the compositional analysis.

Precisely, we consider the pattern in figure 4.11. In this particular pattern, `a = a0;` means `a = A()` i.e. **a** is initialized independently of the input (with the value a_0). The stopping condition for the loop is given by the Boolean valued expression `S(i,a)`.

```
ListFoo(i){
      a = a0;
      b = b0;
      while(! S(i,a)){
            c = O(a,b,c,i);
            l.add = c;
            a = M(a,c);
            b = N(i,b,c); }
      return l; }
```

Figure 4.12: Collecting **c** values in a list

We shall now prove that a program having the generic structure of *foo* given in Figure 4.11 has, under certain conditions, the property $P_{k,\varepsilon,\delta}$ for some k, ε, δ.

4.3.2 A sufficient condition for robustness

Once a program matches the template in figure 4.11, the schema variables `O`, `M` and `N` will be bound to fragments of the original program. We will now construct new, related programs using the values bound to `O`, `M` and `N`.

The first such program in figure 4.12 extracts the list of the values of **c** during the execution of the list. For instance, if, when *foo* is executed with input **i**, the loop is executed 3 times and the value of **c** is 1 at the first iteration then 2 and 3, then *listFoo*, executed with the same input **i**, returns the list $\{1,2,3\}$.

In this pseudo code, we use the notation `l.add=c;` to denote the addition of the element **c** to the end of the list **l** which is initially empty. This line is not part of the initial code but can be implemented in any language: the format of that list just needs to be consistent with the rest of the programs.

We now define two new programs. The first one is the foo_b program given in figure 4.13: it has the same shape as *foo* but instead of setting **c** by the computation of `O(a,b,c,i)`, it sets **c** with the values of a list given in input. Naturally, the stop condition for the loop is now that all elements of the list have been accessed. Note that, since **a** was just used in the computation of `O`, the commands affecting **a** are now useless and can be removed. They have been commented with the `//` syntax.

Here, we have used Java-style instructions such as `l.length` for the length of the list **l** and `l[j]` for the j^{th} element of the list **l**. The foo_b program, by its structure, allows us to do a dissociation between the variations due to the control flow and the ones coming from the computations themselves : we define a new function $foo_B(i,i') = foo_b(listFoo(i),i')$. This function

```
foo_b(l,i){
//    a = a0;
      b = b0;
      for(int j = 0; j < l.length; j++ ){
            c = l[j];
//          a = M(a,c);
            b = N(i,b,c); }
      return b; }
```

Figure 4.13: The foo_b program

```
foo_a(l){
      a = a0;
//    b = b0;
      for(int j = 0; j < l.length; j++ ){
            c = l[j];
            a = M(a,c);
//          b = N(i,b,c);
      }
      return a; }
```

Figure 4.14: The pattern of foo_a

mixes the control flow that would appear with input i but does computations with inputs i'. In particular, $foo_B(i,i) = foo(i)$.

The second program $foo_a(l)$ is the same program as foo_b except that **a** is returned instead of **b**. In this program, the lines where **b** is set are now useless. When l contains the first k elements of $listFoo(i)$, $foo_a(l)$ returns the value of **a** after k iterations of the execution of the loop in $foo(i)$. For readability reasons, we define $foo_A(i) = foo_a(listFoo(i))$: this value correspond to the final state of the variable **a** when *foo* is executed with input i. In particular, $S(i, foo_a(i))$ is true.

We now introduce four conditions that need to hold to prove that the *foo* program satisfies the $P_{k,\varepsilon,\delta}$ property for appropriate values of k, ε, and δ. Condition 4.3.1 expresses the property $P_{k_{N^*},\varepsilon_{N^*},\delta}$ for the transformed program foo_B, condition 4.3.2 expresses the fact that there is a relationship between the values stored in **a** and the values stored in **b**, and conditions 4.3.3 and 4.3.4 address the stability of the stop condition `S(i,a)`.

Condition 4.3.1.

$$\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z)).$$

The next condition expresses that when the program foo_A returns similar results with some

arguments i and i', it is also the case for the program foo_B.

Condition 4.3.2.

$$\forall i_1, i \in I, d_i(i, i_1) \leq \delta \implies d_b(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_a(foo_A(i_1), foo_A(i)) + \varepsilon_2$$

The stopping condition S should satisfy the following two conditions. The first expresses that the boundary of the region $\{a \mid S(i,a)\}$ cannot vary too much.

Condition 4.3.3.

$$\forall a \in A, \forall i, i' \in I, d_i(i,i') \leq \delta \wedge S(i',a) \implies \exists a' \in A, d_a(a,a') \leq k_s d_i(i',i) + \varepsilon_s \wedge S(i,a')$$

The following condition on S states that the diameter of the region $\{a \mid S(i,a)\}$ is as small as the desired precision.

Condition 4.3.4.

$$\forall a, a' \in A, \forall i \in I, S(i,a) \wedge S(i,a') \implies d_a(a,a') \leq \varepsilon_t$$

Finally, our main theorem is the following.

Theorem 4.3.1. *If the program foo terminates and the four conditions hold, then $P_{k_0,\varepsilon_0,\delta}$ holds for the function computed by foo with $k_0 = k_{N^*} + k_A k_s$ and $\varepsilon_0 = \varepsilon_{N^*} + k_A(\varepsilon_s + \varepsilon_t) + \varepsilon_2$.*

Proof. In the proof, we will use these two observations:

1. Since $listFoo(i)$ is obtained from the computation of $foo(i)$, and since $foo_B(i,i')$ replaces the result of O by this list, if we compute $foo_B(i,i)$ we are replacing each value for c by itself. Therefore we have that $foo(i) = foo_B(i,i)$.

2. In the execution of $foo(i)$, the final value of a that satisfies the stopping condition $S(i,a)$ is $foo_A(i)$.

By the observation 1, proving the theorem is equivalent to proving

$$\forall i, i0 \in I, d_i(i,i0) \leq \delta \implies d_b(foo_B(i,i), foo_B(i0,i0)) \leq k_0 d_i(i,i0) + \varepsilon_0.$$

By condition 4.3.1, choosing $l = listFoo(i0)$, we have

$$\forall i, i0 \in I, d_i(i,i0) \leq \delta \implies d_b(foo_b(listFoo(i0), i0), foo_b(listFoo(i0), i)) \leq k_{N^*} d_i(i,i0) + \varepsilon_{N^*}.$$

By definition of foo_B, we have

$$\forall i, i0 \in I, d_i(i,i0) \leq \delta \implies d_b(foo_B(i0,i0), foo_B(i0,i)) \leq k_{N^*} d_i(i,i0) + \varepsilon_{N^*}. \tag{4.1}$$

From observation 2, $S(i0, foo_A(i0))$ holds. By condition 4.3.3 (instantiating i' with $i0$) we derive that:

$$\forall i, i0 \in I, d_i(i,i0) \leq \delta \implies \exists a' \in A, d_a(foo_A(i0), a') \leq k_s d_i(i,i0) + \varepsilon_s \wedge S(i,a'). \tag{4.2}$$

Hence, by observations 2 and 1, $S(i, foo_A(i))$ also holds. From inequality (4.2) and condition 4.3.4, we derive

$$d_a(a', foo_A(i)) \leq \varepsilon_t. \tag{4.3}$$

From the last inequality and from inequality (4.2), we derive, using the triangle inequality

$$d_a(foo_A(i0), foo_A(i)) \leq k_s d_i(i,i0) + \varepsilon_s + \varepsilon_t. \tag{4.4}$$

From condition 4.3.2 and inequality (4.4), we have

$$\forall i, i0 \in I, d_i(i,i0) \leq \delta \implies d_b(foo_B(i0,i), foo_B(i,i)) \leq k_A(k_s d_i(i,i0) + \varepsilon_s + \varepsilon_t) + \varepsilon_2. \tag{4.5}$$

From inequalities (4.1) and (4.5), using the triangle inequality, we derive

$$\begin{gathered} \forall i, i0 \in I, d_i(i,i0) \leq \delta \\ \implies \\ d_b(foo_B(i,i), foo_B(i0,i0)) \leq k_{N^*} d_i(i,i0) + \varepsilon_{N^*} + k_A(k_s d_i(i,i0) + \varepsilon_s + \varepsilon_t) + \varepsilon_2. \end{gathered}$$

Finally, we define $\varepsilon_0 = \varepsilon_{N^*} + k_A(\varepsilon_s + \varepsilon_t) + \varepsilon_2$ and $k_0 = k_{N^*} + k_A k_s$.

□

4.3.3 Application to the CORDIC algorithm

We now illustrate how the theorem applies with the implementation of the CORDIC algorithm 4.1.1.

Our proof is not done for the code 4.2 given in 4.1.1 but instead on the code in 4.15. The only change to the CORDIC code is that the stopping condition has been replaced by an equivalent one.

Now, we prove this implementation is $P_{k,\varepsilon,\infty}$. We do not compute the exact constants since they depend on the analysis of foo_B which can be optimized much better than with a "by hand"

```
double cos(double beta)
{
    double x = 1, y = 0, x_new, theta = 0, sigma, e = 1E-10;
    int Pow2 = 1;
    while(|theta - beta| > e) {
         Pow2 *= 2;
         if(beta > theta)
           sigma = 1;
         else
           sigma = -1;
         sigma = sigma / Pow2;
         theta = theta + atan(sigma); // Value stored
         fact= cos(atan(sigma)); // Value stored
         x_new = x + y * sigma;
         y = fact * (y - x * sigma);
         x = fact * x_new; }
    return x; }
```

Figure 4.15: Another implementation of the CORDIC algorithm

proof.

Scheme instantiation

To apply our method, we have first of all to instantiate the schema variables **a**, **b**, **c** (cf. Section 4.3.2) with a suitable partition of the variables of the program. The variables in **i** are instantiated with the variables which represent the input.

In this example, the partition for the variables will be the following.

```
a := double theta;
b := double x,y;
c := double sigma;
i := double beta;
```

We now must define a suitable metric on the types of the variables in **a** and **b**. We choose the following:

- d_a is the usual distance on $\mathbb{R}$.
- d_b is the L_2 norm on $\mathbb{R}^2$.

Then, we need to instantiate the functions `M(a,c)`, `N(i,b,c)`, `O(a,b,c,i)` of the schema with suitable regions of code. We choose these as it follows.

```
O(theta,<x,y>,sigma,beta) {
    Pow2 = 2 * Pow2;
    if(beta > theta)
      sigma = 1;
    else
      sigma = -1;
    sigma = sigma / Pow2;
    return sigma; }

M(theta,sigma) {
    theta = theta + atan(sigma);
    return theta; }

N(beta,<x,y>,sigma) {
    fact = cos(atan(sigma));
    x_new = x + y * sigma;
    y = fact * (y - x * sigma);
    x = fact * x_new;
    return <x,y>; }
```

Finally, we need to prove that the conditions 4.3.1, 4.3.2, 4.3.3 and 4.3.4 of section 4.3.2 are satisfied.

Proofs of the conditions

Condition 4.3.1

$$\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z))$$

To prove this condition we have to analyze the following code that corresponds to foo_b.

```
double cos(double beta, int[] listFoo)
{
    double x = 1, y = 0, x_new, theta = 0, sigma, e = 1E-10;
    for(int j=0;j<listFoo.length;j++) {
         sigma=listFoo[j];
         theta = theta + atan(sigma); // Value stored
         fact = cos(atan(sigma)); // Value stored
         x_new = x + y * sigma;
```

```
        y = fact * (y - x * sigma);
        x = fact * x_new; }
  return x;
}
```

As we have explained, the main goal of our method is to transform a code into a simpler code which does not have branches anymore. Since this code contains only simple instructions and the domain of the variables remain bounded, it is staightforward to see it is $P(k_{N^*}, \varepsilon_{N^*})$ for some k_{N^*} and ε_{N^*}. The exact computation of k_{N^*} and ε_{N^*} is not the purpose of this example however.

Condition 4.3.2

$$\forall i_1, i \in I, d_i(i, i_1) \leq \delta \implies d_b(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_a(foo_A(i_1), foo_A(i)) + \varepsilon_2$$

The proof of condition 4.3.2, is the most difficult part of this example. We have proved it "by hand", and we do not claim that there is an easy way to automate it. More precisely, it is the difficulty of this proof that motivates us to find another method which is presented in the next section. However, this proof points out that we can prove the intended property without considering the whole semantics of the program, but just the relevant properties.

We start by observing that our program satisfies the following properties.

1. There exists ε_M^-, such that $\forall a \in A, k_M|c - c'| - \varepsilon_M^- \leq d_a(M(a,c), M(a,c'))$
2. For all $i \in I, \lambda x N(i, b0, x)$ is $P_{k_N, \varepsilon_N^+, \infty}$.
3. For all $c \in C$, $\lambda a M(a,c)$ is $P_{1, \varepsilon_M, \infty}$.
4. For all $i \in I, c \in C$, $\lambda b N(i,b,c)$ is $P_{1, \varepsilon'_N, \infty}$.
5. $$\exists \varepsilon_{N^2}, \forall z, c \in [0, \pi], d_b(N(i, N(i, b0, z), c), N(i, b0, (f(z,c)))) \leq \varepsilon_{N^2}$$
   where $f(z,c) = \tan(\text{atan}(z) + \text{atan}(c))$
6. $$\exists \varepsilon_{M^2}, \forall z, c \in [0, \pi], d_a(M(M(a0, z), c), M(a0, (f(z,c)))) \leq \varepsilon_{M^2}$$
7. The number s of loop iterations is fixed.

The observation 1 is kind of reciprocal of the $P(k,\varepsilon)$ property. The constants can be found with the same compositional techniques.

The observations 2, 3 and 4 that some functions are $P_{k,\varepsilon}$ for some k and ε is straightforward since all these functions are composition of basic $P_{k,\varepsilon}$ function like addition and multiplication on a bounded domain. The exact computation of these constants have to be done automatically: there is no interest in computing them by hand here.

The observations 5 and 6 are closeness relationship between the composition of `M` and `N`, respectively, and some mathematical functions. To obtain these epsilons, an analyzer has to compute the closeness of all atomic operation and use the weak compositionality property.

The observation 7 is true whenever the parameter e of the stopping condition is not too small, this problem have been detail in 4.1.1 on the paragraph about figure 4.5.

We will now prove the following generalization of Condition 4.3.2:

$$\forall i, i_1 \in I, d_b(foo_B(i,i), foo_B(i,i_1) \leq k_A d_a(foo_A(i), foo_A(i_1)) + \varepsilon_2$$

We start by proving the following lemma.

Lemma 4.3.1.

$$\forall l \in \mathbb{R}^n, \forall i \in I, \exists z \in \mathbb{R}, d_b(N(i,b0,z)), foo_b(l,i)) \leq \varepsilon_B \wedge d_a(M(a0,z), foo_a(l)) \leq \varepsilon_A$$

Proof. This lemma is proved by induction on the size n of a list l. The initial case where the list is empty holds for $z = 0$.

For the general case, we assume the property proved for any list of size n and we prove it for the size $n+1$. First, we have $foo_b(l:c,i) = N(i, foo_b(l,i), c)$. The induction hypothesis gives us:

$$\forall l \in \mathbb{R}^n, \forall i \in I, \exists z \in \mathbb{R}, d_b(N(i,b0,z), foo_b(l,i)) \leq n\varepsilon_B \wedge d_a(M(a0,z), foo_a(l)) \leq n\varepsilon_A$$

Where $\varepsilon_B = \varepsilon_N + \varepsilon_{N^2}$ and $\varepsilon_B = \varepsilon_M + \varepsilon_{M^2}$.

Let call $z_l \in \mathbb{R}$ be the real provided by our induction hypothesis for our list l. So we have

$$d_b(N(i,b0,z_l), foo_b(l,i)) \leq n\varepsilon_B$$

Now using our observation 3, we derive:

$$d_b(N(i,N(i,b0,z_l),c), N(i, foo_b(l,i),c)) \leq d_b(N(i,b0,z_l), foo_b(l,i)) + \varepsilon_N$$

Hence, from the two last inequalities:

$$d_b(N(i,N(i,b0,z_l),c),N(i,foo_b(l,i),c)) \leq \varepsilon_B + \varepsilon_N$$

By a triangular inequality using observation 5 and the last inequality, we derive:

$$d_b(N(i,b0,(f(z_l,c))),N(i,foo_b(l,i),c)) \leq n\varepsilon_B + \varepsilon_N + \varepsilon_{N^2}$$

The same steps for M (we use observation 4) ends at:

$$d_a(M(a0,(f(z_l,c)),M(foo_a(l),c)) \leq n\varepsilon_A + \varepsilon_M + \varepsilon_{M^2}$$

Finally we have

$$\forall l \in \mathbb{R}^{n+1}, \forall i \in I, \exists z \in \mathbb{R},\ d_b(N(i,b0,z),foo_b(l,i)) \leq (n+1)\varepsilon_B \wedge d_a(M(a0,z),foo_a(l)) \leq (n+1)\varepsilon_A$$

□

We can now complete the proof of condition 4.3.2.

Proof. From the first inequality of Lemma 4.3.1 (we get from observation 7 that n can be bound by s) and triangular inequalities, we have

$$\forall l_1,l_2 \in C^*, i \in I, \exists z_1,z_2 \in \mathbb{R}, d_b(foo_b(l_1,i),foo_b(l_2,i)) \leq d_b(N(i,b0,z_1),N(i,b0,z_2)) + 2s\varepsilon_B$$

By using observation 2, we have

$$d_b(N(i,b0,z_1),N(i,b0,z_2)) \leq k_N|z_1 - z_2| + \varepsilon_N^+$$

Because of observation 1, we have

$$|z_1 - z_2| \leq k_M d_a(M(a0,z_1),M(a0,z_2)) + \varepsilon_M^-$$

From the three last inequalities we get

$$\forall l_1,l_2 \in C^*, i \in I, d_b(foo_b(l_1,i),foo_b(l_2,i)) \leq (k_N(k_M d_a(M(a0,z_1),M(a0,z_2)) + \varepsilon_M^-) + \varepsilon_N^+ + 2s\varepsilon_B$$

By using the second part of our lemma we have

$$d_a(M(a0,z_1),foo_a(l_1)) \leq s\varepsilon_A$$

and

$$d_a(M(a0,z_2),foo_a(l_2)) \leq s\varepsilon_A$$

Finally, the triangle inequality between the three last inequalities, allows us to derive:

$$\forall l_1,l_2 \in C^*, i \in I, d_b(foo_b(l_1,i),$$

$$foo_b(l_2,i)) \leq (k_N(k_M(d_a(foo_a(l_1),foo_a(l_2)+2s\varepsilon_A))+\varepsilon_M^-)+\varepsilon_N^+ + 2s\varepsilon_B$$

Hence we have $k_A = k_N k_M$ and $\varepsilon_2 = 2k_N k_M s\varepsilon_A + k_N\varepsilon_M^- + \varepsilon_N^+ + 2s\varepsilon_B$. □

Condition 4.3.3

$$\forall a \in A, \forall i,i' \in I, d_i(i,i') \leq \delta \wedge S(i',a) \implies \exists a' \in A, d_a(a,a') \leq k_s d_i(i',i) + \varepsilon_s \wedge S(i,a')$$

Proof. The instantiation of $S(i,a)$ corresponds to $|i-a| \leq e$, so condition 4.3.3 is given by the condition:

$$\forall a \in A, \forall i,i' \in I, |i-a| \leq e, \exists a' \in I, |a-a'| \leq k_s|i-i'| + \varepsilon_s \wedge |i'-a'| \leq e$$

We can satisfy this property by setting $a' = a + i' - i$, $k_s = 1$, and $\varepsilon_s = 0$. □

Condition 4.3.4

$$\forall a,a' \in A, \forall i \in I, S(i,a) \wedge S(i,a') \implies d_a(a,a') \leq \varepsilon_t$$

Proof. Condition 4.3.4 can be rewritten, once we instantiate $S(i,a)$ to

$$\exists \varepsilon_t, \forall a,a' \in A, \forall i \in I, |i-a| \leq e \wedge |i-a'| \leq e \implies |a-a'| \leq \varepsilon_t$$

Which is true for $\varepsilon_t = 2e$. □

4.3.4 Application to the Dijkstra's shortest path algorithm

Now, we apply our method to Dijkstra's shortest path algorithm 4.1.2. We prove by instantiating our schema that the Dijkstra's algorithm is $P(1,\varepsilon,\delta)$ for some ε depending on the finite representation on the input domain where all edge have a greater value than δ. The restriction on the domain is a weakness of this method as we will detail further.

Scheme instantiation

To apply our theorem, we have to instantiate the abstract variables **a**, **b** and **c** with some variables of the program. The abstract variable **i** are instantiated with the variables that represent the input. We choose the following instantiation: **a** contains the variables *count* and *mark*, **b** the array of double *pathestimate* and **c** the variable `u` which identifies the current vertex to propagate.

```
A := int count;int mark[w];
B := pathestimate[w];
C := int u;
I := graph[w][w];
```

We, now, have to choose a suitable metric on the types of the variables, and we choose the following: d_i is the L_1 norm on an array of real numbers, d_b is the L_∞ norm on array of real numbers and d_a is the identity metric: that is, the distance between two elements of A is 0 if they are the same elements and it is ∞ otherwise.

Next, we identify the stopping condition:

```
S(graph,<count,mark>) := count >= w
```

Finally, we identify the functions $M(a,c)$, $N(i,b,c)$, $O(a,b,c,i)$ with the following regions of code:

```
O (count, mark, pathestimate, u, graph) {
   u=minimum(pathestimate,mark,w);
   int minimum(int a[],int m[],int k){
        int mi=999;
        int i,t;
        for(i=1;i<=k;i++){
             if(m[i]!=1){
                  if(mi>=a[i]){
                       mi=a[i];
                       t=i;
                  }
             }
        }
       return t;
   }
   return u;
```

```
}

M (<mark, count>, u) {
   mark[u]=1;
   count=count+1;
   return <mark,count>;
}

N (graph, pathestimate, u) {
   for(i=1;i<=w;i++){
        if(pathestimate[i]>pathestimate[u]+graph[u][i]){
             pathestimate[i]=pathestimate[u]+graph[u][i];
        }
   }
   return pathestimate;
}
```

We, now, have to prove that the four conditions hold for the given instantiation.

Proof of the conditions

Condition 4.3.1

$$\forall l \in C^*.P_{k_{N^*},\varepsilon_{N^*},\delta}(\lambda z.foo_b(l,z))$$

For all $i0 \in I$, $foo_a(i0,i)$ is k-Lipschitz and k does not depend on $i0$. The proof of this condition can be done by using standard technical (such as Hoare triples or abstract interpretation) on the following program.

```
int[] dijkstra( int graph[w][w], int[] listFoo)
{
    int pathestimate[w],mark[w];
    int source,i,j,u,predecessor[w],count=0;
    int minimum(int a[],int m[],int k);
    for(j=1;j<=w;j++){
        mark[j]=0;
        pathestimate[j]=999;
        predecessor[j]=0;
    }
```

```
    source=0;
    pathestimate[source]=0;
    for(j=0;j<listFoo.length;j++){
        u=listFoo[j];
        for(i=1;i<=w;i++) {
            if(pathestimate[i]>pathestimate[u]+graph[u][i]){
                pathestimate[i]=pathestimate[u]+graph[u][i];
                predecessor[i]=u;
            }
        }
    }
    return pathestimate;
}
```

Condition 4.3.2

$$\forall i_1, i \in I, d_i(i,i_1) \leq \delta \implies d_b(foo_B(i,i), foo_B(i_1,i)) \leq k_A d_a(foo_A(i_1), foo_A(i)) + \varepsilon_2$$

Since d_a is the identity metric, the condition 4.3.2 can be rewritten as

$$\exists \varepsilon_2 \in \mathbb{R}, i, i_1 \in I, d_i(i,i_1) \leq \delta \implies foo_A(i_1) = foo_A(i) \implies d_b(foo_B(i,i), foo_B(i_1,i)) \leq \varepsilon_2$$

Like for the CORDIC algorithm, condition 4.3.2 is rather difficult to prove. Here, there is an additional problem. Indeed, our program analysis compel us to analyze the program with the control flow of another input. However, in general, this other input can be far away from the original one. More precisely, the order the nodes are processed can be in an arbitrary order: for any order, there exists a graph such that the Dijkstra's algorithm processes the nodes in this order. Since at the end of the program, we always have $foo_A(i_1) = foo_A(i)$, if we take $\delta = \infty$ to prove this condition we have to prove that the order in which we do the computation does not matter. This assertion is false, however. For instance, if an update is made from a node which have never been updated (the value associated to the node is still 999) then the update does not change any other values of neighbor node. This update, however, marks the node as processed preventing the node to propagate values anymore.

So, we need to limit the range of i_1 such that the ensued order is not too different from the initial one. More precisely, we can allow the control flow to select a node which is not the minimal one provided that this node cannot be updated further by a minimal value. This is

possible if the selected value differs from the minimal value of at most the minimal value of any edge in the graph. That is why we introduce δ: the value of δ is a lower bound on edge length.

This limitation is strong since it restricts the domain of validity for the input (no graph with null edge is allowed) and compel us to only prove the $P(k,\varepsilon,\delta)$ property instead of the much more natural $P(k,\varepsilon)$ property.

Proof. Since d_a is the identity metric and by instantiating the variables of the pattern, we can derive from the equality $foo_A(i) = foo_A(i_1)$ that $foo_A(graph)$ and $foo_A(graph_1)$ return the same pair $\langle count, mark\rangle$. In particular, we have these two properties.

- Since *count* is incremented once at each iteration of the loop, $listFoo(i)$ and $listFoo(i_1)$ have the same length.
- Since *mark* sets u to 1 the element , this means $listFoo(i)$ and $listFoo(i_1)$ contains the same values for u (which instantiates c).

So $listFoo(i_1)$ is a permutation of $listFoo(i)$. Moreover, any permutation on a list can be generated by a sequence of transpositions. In our particular case, this means that there exists a list of lists $l_0,\ldots,l_n$ where $l_0 = listFoo(i)$ and $l_n = listFoo(i0)$ such that: l_{p+1} is a transposition $\langle n_p, n_p+1\rangle$ of l_p. A auxiliary analysis about the program $N(graph,N(graph,pathestimate,u),v)$ can provide the following result:

$$\begin{aligned}&\forall u,v\in\mathbb{N}, pathestimate\in\mathbb{R}^n, pathestimate[v]\leq pathestimate[u]+graph[u][v]\\ \Longrightarrow\ & N(graph,N(graph,pathestimate,u),v) = N(graph,N(graph,pathestimate,v),u)\end{aligned}$$

Moreover,

$$pathestimate[v]\leq pathestimate[u]+graph[u][v]$$

implies the values $pathestimate[u]$ and $pathestimate[v]$ does not change while computing $N(graph,N(graph,pathestimate,u),v)$.

Hence, it is sufficient to prove in our particular case,

$$\forall u,v, pathestimate[v]\leq pathestimate[u]+graph[u][v].$$

We will prove the equivalent condition:

$$\forall u,v, |pathestimate[u]-pathestimate[v]|\leq graph[u][v].$$

We denote by $(pathestimate(l,i))$ the list of B whose the j^{th} element is the value $pathestimate[l[j]]$ at the end of the execution of $foo_b(l,i)$.

From the analysis of $O(count, mark, pathestimate, u, graph)$, we can see that it extracts always the minimum of *pathestimate* which are not in *mark* and since $N(graph, pathestimate, u)$ always sets value greater than $pathestimate[u]$, the list $pathestimate(listFoo(i), i)$ is sorted.

Moreover, since the code fragment $N(graph, pathestimate, u)$ always decreases the values of *pathestimate*, the value of $pathestimate[u]$ is not changed from the iteration where $u = O(a,b,c,i)$.

Since $(pathestimate(listFoo(i0), i0)[u])$ is sorted,

$$\forall u \in [0, n-1], pathestimate(listFoo(i0), i0)[u] < pathestimate(listFoo(i0), i0)[u+1].$$

Moreover since $\lambda x foo_b(listFoo(i0), x)$ is $P_{k_{N^*}, \varepsilon_{N^*}}$, $(pathestimate(listFoo(i0), i)$ is almost sorted, that is,

$$\forall u, pathestimate(listFoo(i0), i)[u] - k_{N^*} d_i(i - i0) - \varepsilon_{N^*} < pathestimate[u+1]_i 0 + k_{N^*} d_i(i - i0) + \varepsilon_{N^*}.$$

So, if two elements are not well ordered then they differ by at most $2k_{N^*}|i - i0| + 2\varepsilon_{N^*}$. So, we can permute them without changing the result if

$$2k_{N^*} d_i(i - i0) + 2\varepsilon_{N^*} \leq \delta. \tag{4.6}$$

Since, the list obtained through these transpositions is sorted, we obtain the same list as $(pathestimate(listFoo(i), i)[u])$. So, this new list actually computes $foo(i)$. So, we proved

$$\exists \varepsilon_2 \in \mathbb{R}, i, i_1 \in I, d_i(i, i_1) \leq \delta \wedge foo_A(i) = foo_A(i_1) \implies d_b(foo_B(i,i), foo_B(i_1, i)) \leq 0$$

□

Condition 4.3.3

$$\forall a \in A, \forall i, i' \in I, d_i(i, i') \leq \delta \wedge S(i', a) \implies \exists a' \in A, d_a(a, a') \leq k_s d_i(i', i) + \varepsilon_s \wedge S(i, a')$$

Proof. Since the instantiation of $S(i', a)$ is `count >= w`, the stopping condition does not depend on i (when the number of nodes w is fixed). Hence, the formula is satisfied for $a' = a$ with the constant $k_s = 0$ and $\varepsilon_s = 0$. □

Condition 4.3.4

$$\forall a, a' \in A, \forall i \in I, S(i, a) \wedge S(i, a') \implies d_a(a, a') \leq \varepsilon_t$$

Proof. Since $\{a|S(i,a)\}$ is a singleton for every i (it corresponds to the state where all the nodes are marked), the property holds for $\varepsilon_t = 0$. □

4.3.5 Discussion

In this section, we only consider the finite-precision semantics. The analysis of the program was based on program transformation so that the discontinuity of program does not appear in its transformed version. This approach seemed appealing until we really try to apply it on concrete example. Indeed, the conditions of the theorems aimed at being straightforward to prove, which is not actually the case.

There are two main problems in this method. First, working only with the finite precision semantics induce big over-approximations. Indeed, to instantiate condition 4.3.2 on CORDIC, we had to introduce the exact trigonometric function (observation 5).

Secondly, our program transformation introduces the $foo_b(i,i')$ program. This new function allows us to study the behavior of the program when the control flow is fixed. In return, this function looses information by considering cases that can never happen. This has been illustrated by the Dijkstra's instantiation where we were just able to prove a $P(k,\varepsilon,\delta)$ property while there is no inherent reason to limit the result for a particular δ (smaller than the minimal edge).

To solve these problems, we have developed another solution based one the exact semantics that we present in the next section.

4.4 Analysis through rewriting techniques

The previous section aims at proving the $P(k,\varepsilon)$ property for the finite-precision semantics. This approach is useful for automatic analysis when we just have the code of the program but not its meaning. We tried this approach because working only on the finite precision is standard when programs are quite simple and when the difficulty resides mostly on the size of the code.

However, when people are writing programs, they want a precise property for their program. We can assume, for instance, the program relies on some mathematical property coming from some theorem. In that case, we have additional knowledge: the program is correct in the exact semantics (we assume that the transcription has been done without error). However, we do not know what the program does in the finite-precision semantics. Mostly, we cannot be sure that the control flow in the finite-precision semantics is the same as in the exact semantics.

Indeed, consider the following example:

```
root(a,b,c){
  if( 0.25 * b * b  >= a * c){
```

```
    return (- b + sqrt( b * b - 4 a * c)) / (2 * a);
  else
    return - b / (2 * a);
}
```

From a mathematical point of view, this program always returns the real part of the root of the quadratic aX^2+bX+c. In the finite precision semantics however, this program can fail. Indeed, due to rounding error, we can have the test (`0.25 * b * b >= a * c`) that succeeds while the expression `b * b - 4 a * c` is evaluated into a negative value. Since the square root function cannot handle negative values, a run-time exception is raised.

In this setting, we propose a method that allows to extend a result for exact semantics to finite-precision semantics for the same problem as in the last section: when the continuity of the function just appears once the execution ends while intermediate steps are non robust.

Since, the method is based on the knowledge of the exact semantics, we will be able to get a better property than just the $P(k,\varepsilon)$: we prove the (k,ε)-closeness between the exact and the finite-precision semantics.

4.4.1 Preliminaries: the rewriting framework

We have seen in chapter 2 that the finite precision semantics is never computed but over approximated either by a particular semantics like zonotopes or by studying a property that states the semantics remains in some range. The main idea, when a branch is encounter, consisted in merging the properties of the two branches in to one property. Here, we are interested, in the case, such kind of technique is impossible. Indeed, when the control flow can select a branch which has nothing to do with the other one there is no common denominator. Therefore, after the branch, there is no more information available about the program.

Here, we split the problem into two parts: what happens in one branch is still studied in a classical way, but, here, branching is seen as a non deterministic choice. In this setting, we do not have to merge the properties of all branches, we just analyze all branches then with some additional conditions, we prove that for any branch the result will be almost the same.

In this setting, the problem of a global property which does not hold in intermediate step can be seen as a problem of confluence in the rewriting system theory.

Before looking at how this framework applies, we first present what are rewriting systems. We first present the formal definition about abstract rewrite systems (also called reduction systems).

Definition 4.4.1 (Abstract reduction system)**.** *An abstract reduction system is a pair* $(A,\rightarrow)$ *where the* reduction $\rightarrow$ *is a relation between on the set A, i.e.* $\rightarrow \subseteq A\times A$*. We denote by* $a\rightarrow b$

the property $(a,b) \in \rightarrow$.

Definition 4.4.2 (Transitive closure)**.** *The transitive closure* $\rightarrow^*$ *of* $\rightarrow$ *is the least transitive and reflexive relation that contains* $\rightarrow$.

Definition 4.4.3 (Normal form)**.** *An element a of A is in normal form if there is no reduction* $a \rightarrow b$.

Definition 4.4.4 (Acyclic)**.** *A reduction system is acyclic if there exists no a such that* $a \rightarrow b \rightarrow^* a$.

Definition 4.4.5 (Terminating)**.** *A reduction system is terminating if there is no infinite chain* $a \rightarrow b \rightarrow c \rightarrow \ldots$.

Definition 4.4.6 (Local confluence)**.** *A reduction system is locally confluent if for all a, b and c, we have some d such that*

$$b \leftarrow a \rightarrow c \implies b \rightarrow^* d \ ^* \leftarrow c$$

Definition 4.4.7 (Global confluence)**.** *A reduction system is globally confluent if for all a, b and c, we have some d such that*

$$b \ ^* \leftarrow a \rightarrow^* c \implies b \rightarrow^* d \ ^* \leftarrow c$$

In this section, we only make use of two propositions about reduction systems.

Proposition 4.4.1 ([New42])**.** *In a reduction system which is terminating and globally confluent, all elements a have a unique normal form* $\bar{a}$*, i.e. there exists an unique element in normal* $\bar{a}$ *form such that* $a \xrightarrow{>}{}^* \bar{a}$.

Proposition 4.4.2 (Newman's Lemma [New42])**.** *A terminating and locally confluent reduction system is globally confluent.*

Thus to compute the normal form for a terminating and either local or global confluent reduction, one must repeatedly apply a rewrite rule until no more reduction rules can be applied.

4.4.2 Application of the rewriting framework

The problematic `if` branches that induces local discontinuity can be seen as a choice between different rules we can apply. If we can prove that these rules correspond to a confluent system, then, even if rounding errors change the control flow, the final result will be the same final term. That is the rough idea we develop in this section. We, now, need to formalize it.

The standard way for program to reduce a term to its normal form consists of a `while` loop that proceeds the successive reduction. The stopping condition corresponds to checking whether

the term is in its normal form. In the body of the loop, there are two parts: a first one that selects the reduction rule to apply and the other one that actually does the selected reduction.

This leads us to the following pattern:

```
foo(m){
      (n,i) = Init();
      while(! S(m,n,i)){
       (i,n) = C(m,n,i);
           m = R(i,m);
      }
      return m; }
```

This pattern follows the same conventions as described in section 4.2. Moreover, **m** belongs to $\mathbb{R}^m$ (such that we can match program with several variables), **n** also belongs to $\mathbb{R}^m$ while *i has to belong to a finite set I*.

Remark 3. *This pattern only works for programs of type $\mathbb{R}^m \to \mathbb{R}^m$. However, any program can be transformed in order to fit this requirement: if the input domain In and the output domain Out are different, then we can just set m such that $In \times Out = \mathbb{R}^m$.*

From our former description, the term is represented by the variable **m**, **i** is the number of the selected rule and **n** represent internal variables the scheduler can use. There is four sub-parts of our program: `Init()`, `S`, `C` and `R`. `Init()` just initializes **n** and **i** with some constant values. `S(i,m,n)` is the stopping condition. `C(i,m,n)` is the selector of the rules: it returns the number **i** of the rule to apply and can change its internal state **n**. Finally, `R(i,m)` is the part that applies the selected rewriting rule.

We denote by $\mathcal{R} = \{f_i \mid i \in I\}$ the set of rules of our rewriting system. So $f_i(m) = [\![\texttt{R(i,m)}]\!]$. In the finite precision, the rules do not behave exactly in the same way, we denote by $f_i(m) = [\![\texttt{R(i,m)}]\!]'$ and $\mathcal{R}' = \{f_i' \mid i \in I\}$. Finally, the closure of $\mathcal{R}$ (resp. $\mathcal{R}'$) under function composition is denoted by $\mathcal{R}^*$ (resp. $\mathcal{R}'^*$).

With these definitions, we can define three paths of reductions relative to the computation of *foo* in its exact and finite-precision semantics.

Definition 4.4.8 (Path *a*)**.** *We denote by $a_n(x)$ the value of the variable* **m** *after n executions of the* `while` *loop of foo(x) in the exact semantics. With this notation, we have:*

$$x = a_0(x) \to a_1(x) \to \ldots \to a_n(x) = [\![foo]\!](x)$$

Definition 4.4.9 (Path *c*)**.** *We denote by $c_n(y)$ the value of the variable* `m` *after n executions of*

the `while` *loop of foo*(y) *in the finite-precision semantics. With this notation, we have:*

$$y = c_0(y) \to' c_1(y) \to' \ldots \to' c_n(y) = [\![foo]\!]'(y)$$

Definition 4.4.10 (Path d)**.** *Given $c_0(y) \to' c_1(y) \to' \ldots \to' c_n(y)$, we derive the path. Indeed, when we write $c_j(y) \to' c_{j+1}(y)$ we implicitly refer to the function applied:*

$$c_{j+1}(y) = f'_{i_j}(c_j(y))$$

where i_j is the value of `i` *at the j^{th} iteration of the program. We can define another path by induction: we start with x and we define $d_i \to d_{i+1}$ the reduction made by applying the rule i_j such that*

$$d_{j+1}(x) = f_{i_j}(d_j(x)).$$

We denote by $p(x)$ the final value:

$$x = d_0(x) \to d_1(x) \to \ldots \to d_n(x) = p(x)$$

Our rewrite system is not confluent since there is always a rewriting rule that can be applied to any given $a \in \mathbb{R}^m$.

We are interested in programs that do an iteration on a variable **`m`** and where the stopping condition depends on **`m`**. In this setting, we would like that any term that satisfy the stopping condition to be a final term. However, it is not enough to prevent reduction rules to apply when they reach a final term: if we want to be able to have any kind of confluence property, we need our rewrite system to be acyclic.

To solve this problem, we consider a subsystem of $\to$. This subsystem is designed such that any normal form satisfies the stopping condition `S`. To define such a subsystem, we consider a function $h : \mathbb{R}^m \to \mathbb{R}$ that measures the progress of the reduction. Our subsystem is then defined by removing all rules $a \to b$ such that $h(a) \leq h(b)$. In this setting, a normal form is an element a such that $h(a) < h(b)$ for any $b \neq a$ with $a \to b$.

Definition 4.4.11. *Let $h : \mathbb{R}^m \to \mathbb{R}$ and $\to$ a rewrite system where elements belongs to $\mathbb{R}^m$, we define the rewrite system $\xrightarrow{>}$ with the same elements and the subset of relations $a \to b$ that satisfy: $h(a) > h(b)$.*

We also denote by $a \xrightarrow{>}{}^* b$ the transitive closure of $\xrightarrow{>}$.

Finally, we denote by $\bar{\mathbb{R}^m}$ the set of normal forms in $\mathbb{R}^m$ with respect to the relation $\xrightarrow{>}$, i.e. $\bar{\mathbb{R}^m} = \{m \in \mathbb{R}^m \mid \nexists m' \in \mathbb{R}^m.\ m \xrightarrow{>} m'\}$.

To illustrate how works this rewriting sub system, we propose a toy example.

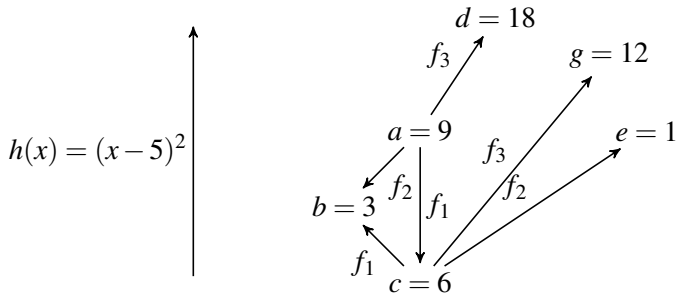

Figure 4.16: A simple rewrite system

Example 4.4.1. *Assume that the elements of our rewrite system are real numbers and that our reduction system $\to$ consists of three rules: $f_1(x) = x-3$, $f_2(x) = x/3$ and $f_3(x) = 2x$. The h function is defined with $h(x) = (x-5)^2$.*

In figure 4.16, we represent the reductions from $a = 9$ and $c = 6$. To express that a reduction belongs to $\xrightarrow{>}$, we represent the elements x at a height that corresponds to $h(x)$. That way, if the arrow $x \to y$ goes down it means that $h(x) > h(y)$ and so $x \xrightarrow{>} y$.

On this picture, we can see that c is in a normal form.

4.4.3 Sufficient conditions to prove the closeness property

To prove the (k,ε)-closeness property between $[\![foo]\!]$ and $[\![foo]\!]'$ by considering the rewriting framework, we have first to match *foo* with the pattern in section 4.4.2 then we have to check the following conditions. The first three conditions are concerned with the rewriting system.

Condition 4.4.1. *$\xrightarrow{>}$ is locally confluent.*

Condition 4.4.2. *The rewriting system $\xrightarrow{>}$ is terminating.*

Condition 4.4.3. *The following property holds.*

$$\forall a,b \in \mathbb{R}^m, a \to b \implies \exists c \in \mathbb{R}^m a \xrightarrow{>}{}^* c \,{}^* \xleftarrow{<} b$$

The next conditions are about the exact semantics. The forth one asks the "smoothness" of the exact function and the fifth one requires that the code `C` terminates to an element close to its normal form.

Condition 4.4.4. *The function $[\![foo]\!]$ is $P(k_e,\varepsilon_e)$.*

Condition 4.4.5. *In the exact semantics, when the stopping condition is reached, the final value m is such that $m \xrightarrow{>}{}^* z$ implies $d(m,z) \leq \varepsilon_s$.*

In addition of these constraints on the exact semantics, there are conditions to grant that the finite-precision semantics is not too far away from the exact semantics. The sixth one is about the closeness between the idealized semantics and the finite-precision semantics and the seventh one is about the termination in the finite-precision semantics.

Condition 4.4.6. *We require the closeness property between $[\![foo]\!]'$ and p (from definition 4.4.10):*

$$\forall x,y \in \mathbb{R}^m, d(p(x),[\![foo]\!]'(y)) \leq k_f d(x,y) + \varepsilon_f.$$

Remark 4. *Depending on the exact code of the program we can get some properties about the possible paths over the approximate semantics. For instance, we might be able to bound the number of iterations. This will allow us to consider a subset of $\mathcal{R}'^*$ otherwise, since $\mathcal{R}'^*$ contains arbitrary long path, there is no chance the closeness property holds for any path of $\mathcal{R}'^*$.*

Condition 4.4.7. *In the finite-precision semantics, when the stopping condition is reached, the final value m' is such that*

$$\exists z' \in \bar{\mathbb{R}^m}, d(m',z') \leq \varepsilon'_s$$

Remark 5. *By reading these condition, there is no mention of the finite precision semantics of the code of C. However, in most cases, if the finite precision semantics of the code of C is completely unsafe, there is no chance that our program is close to the exact semantics.*

In fact, the finite precision semantics of the code of C is implicitly mentioned in condition 4.4.6 and condition 4.4.7. Indeed, either the maximal number of iteration is granted by the stop condition in which case condition 4.4.6 does not need to refer to the code C but condition 4.4.7 is proved through the analysis of code C, or condition 4.4.7 is granted by the stopping condition but condition 4.4.6 should be proved through an analysis of code C to show the program terminates in a bounded number of operations.

Finally, our main theorem is the following.

Theorem 4.4.1. *If all the conditions are satisfied, the function $[\![foo]\!]'$ computed in a finite-precision semantics and the function $[\![foo]\!]$ that would be computed in the idealized semantics are (k,ε)-close (with $k = k_e$ and $\varepsilon = k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 2\varepsilon_s + \varepsilon'_s$) i.e.*

$$\forall x,y \in \mathbb{R}^m, d([\![foo]\!](x),[\![foo]\!]'(y)) \leq k_e d(x,y) + \varepsilon$$

Before starting the proof, we state some useful lemmas.

Proposition 4.4.3. *The rewrite system $\xrightarrow{>}$ is globally confluent.*

Proof. The rewrite system $\overset{>}{\rightarrow}$ is terminating according to Condition 4.4.2 and locally confluent according to Condition 4.4.1, hence, according to the Newman lemma 4.4.2, the rewrite system is globally confluent:

$$\forall a,b,c \in \mathbb{R}^m, b \ ^* \overset{<}{\leftarrow} a \overset{>}{\rightarrow} *c \implies \exists d \in \bar{\mathbb{R}^m}, b \overset{>}{\rightarrow} *d \ ^* \overset{<}{\leftarrow} c$$

□

Proposition 4.4.4. *All elements $x \in \mathbb{R}^m$ have a unique normal form $\bar{x}$.*

Proof. Our rewrite system is terminating by Condition 4.4.2 and globally confluent from proposition 4.4.3, then from proposition 4.4.1 any element $a \in \mathbb{R}^m$ has a unique normal form $\bar{a}$. □

Proposition 4.4.5. *The following assertion, which extends the global confluence property, holds.*

$$\forall x,a,b \in \mathbb{R}^m, b \ ^* \leftarrow x \rightarrow^* a \implies \exists d \in \bar{\mathbb{R}^m}, b \overset{>}{\rightarrow}^* d \ ^* \overset{<}{\leftarrow} a$$

Proof. Let $x \rightarrow x_1 \rightarrow x_2 \rightarrow \ldots x_n$ be a rewrite sequence. We proceed by induction on n. The base case is $n = 0$. Thus x has the same normal form as $\bar{x}$.

Now, we assume that x and x_i have the same normal form.

We apply Condition 4.4.3 on $x_i \rightarrow x_{i+1}$, we get

$$\exists d, x_i \overset{>}{\rightarrow}^* d \ ^* \overset{<}{\leftarrow} x_{i+1}$$

and, so, the normal form of x_i and x_{i+1} are the same.

From our induction hypothesis, we derive that the normal form of x and x_{i+1} is $\bar{x}$.

Finally, we get:

$$\forall x, a \in \mathbb{R}^m, x \rightarrow^* a \implies x \overset{>}{\rightarrow}^* \bar{x} \ ^* \overset{<}{\leftarrow} a$$

By applying this result twice ($b \ ^* \leftarrow x \rightarrow^* a$), we get our proposition. □

Definition 4.4.12 (Path b)**.** *From proposition 4.4.5, we also have the existence of a path that starts from x to $\bar{x}$ using only $\overset{>}{\rightarrow}$ rules.*

$$x = b_0(x) \overset{>}{\rightarrow} b_1(x) \overset{>}{\rightarrow} \ldots \overset{>}{\rightarrow} b_{n'}(x) = \bar{x}$$

This path and the three previous ones are represented in figure 4.17.

Now, we can prove that $[\![foo]\!]'$ and $[\![foo]\!]$ are (k,ε)-close.

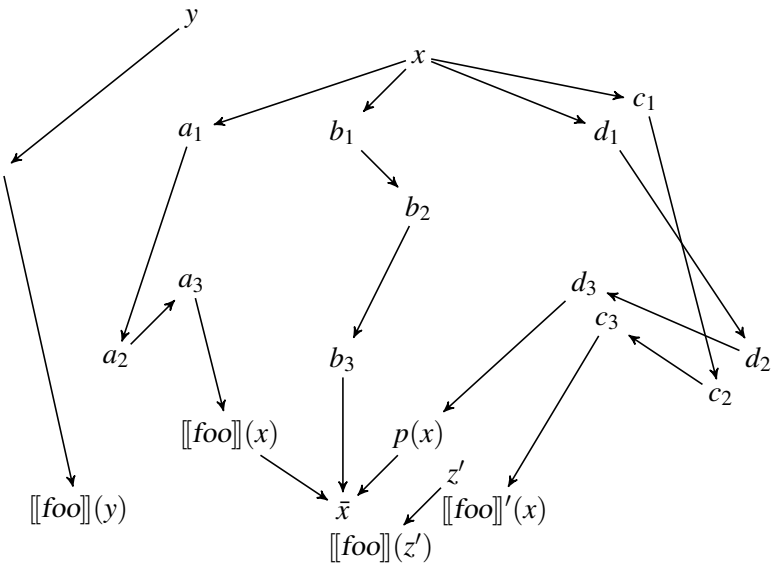

Figure 4.17: The elements introduced in the proof.

Proof. Let $x, y \in \mathbb{R}^m$, we have to prove there exist k and ε such that

$$d([\![foo]\!](y), [\![foo]\!]'(x)) \leq kd(y,x) + \varepsilon.$$

From a triangular inequality, we get

$$d([\![foo]\!](y), [\![foo]\!]'(x)) \leq d([\![foo]\!](y), [\![foo]\!](x)) + d([\![foo]\!](x), [\![foo]\!]'(x))$$

From condition 4.4.4, we have

$$d([\![foo]\!](y), [\![foo]\!](x)) \leq k_e d(x,y) + \varepsilon_e. \tag{4.7}$$

So, now, we have to find a bound for $d([\![foo]\!](x), [\![foo]\!]'(x))$.

We have proved the $\xrightarrow{>}$ rewriting system to be normalizing (proposition 4.4.4), so there exists a unique normal form $\bar{x}$ for x. Moreover, we can derive from the property 4.4.5 $[\![foo]\!](x) \xrightarrow{>} {}^*\bar{x}$. Then, condition 4.4.5 states that $d([\![foo]\!](x), \bar{x}) \leq \varepsilon_s$.

By definition of the path d, we have $x \to^* p(x)$. Since $x \to^* p(x)$ and $x \xrightarrow{>}{}^* \bar{(}x)$, we derived from the property 4.4.5:

$$\exists d \in \bar{\mathbb{R}^m}, p(x) \xrightarrow{>}{}^* d \, {}^* \xleftarrow{<} \bar{(}x).$$

Then, since $\bar{(}x) \in \bar{\mathbb{R}^m}$, $d = \bar{(}x)$ and so:

$$p(x) \xrightarrow{>}{}^{*} \bar{(}x). \tag{4.8}$$

Now, Condition 4.4.7 provides some $z' \in \bar{\mathbb{R}^m}$ with $d([\![foo]\!]'(x), z') \leq \varepsilon_s$. From this inequality and the inequality of the Condition 4.4.6, we derived with a triangular inequality: $d(z', p(x)) \leq k_f d(x,x) + \varepsilon_f + \varepsilon_s$ And then:

$$d(z', p(x)) \leq \varepsilon_f + \varepsilon_s. \tag{4.9}$$

Moreover, since $[\![foo]\!]$ computes a value close to the normal form and by using equation 4.8, we have

$$d([\![foo]\!](p(x)), p(x)) \leq \varepsilon_s \tag{4.10}$$

Also, since z is a normal form, we have

$$d([\![foo]\!](z'), z') \leq \varepsilon_s. \tag{4.11}$$

Condition 4.4.4 on $p(x)$ and z:

$$d([\![foo]\!](p(x)), [\![foo]\!](z')) \leq k_e d(p(x), z') + \varepsilon_e$$

become the following by using equations 4.10 and 4.11.

$$d([\![foo]\!](x), z') \leq k_e d(p(x), z') + \varepsilon_e + 2\varepsilon_s.$$

Then, by using the inequality 4.9, we get:

$$d([\![foo]\!](x), z') \leq k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 2\varepsilon_s \tag{4.12}$$

Now, we do a triangular inequality from inequalities 4.12, 4.7 and condition 4.4.7:

$$d([\![foo]\!](y), [\![foo]\!]'(x)) \leq k_e d(x,y) + \varepsilon_e + k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 2\varepsilon_s + \varepsilon'_s.$$

Therefore $[\![foo]\!]$ and $[\![foo]\!]'$ are $(k_e, k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 2\varepsilon_s + \varepsilon'_s)$-close. □

4.4.4 Application to the CORDIC algorithm

Here, we explain how to apply our method to the CORDIC algorithm presented in section 4.1.1. We prove the exact and the finite-precision semantics are (k,ε)-close for some computable k and ε parameters. We do not prove the presented code 4.2 but a slightly different version (see figure

```
1  double cos(double beta) {
2      double x = 1, y = 0, theta = 0, x_new, sigma;
3      int n = 15, i;
4      for( i = 1 ; i < n ; i++ ) {
5           if( beta > theta )
6             sigma = 1;
7           else
8             sigma = -1;
9           sigma =  sigma / (1 << i);
10          theta = theta + atan(sigma); // Value stored
11          fact = cos(atan(sigma)); // Value stored
12          x_new = x + y * sigma;
13          y = fact * (y - x * sigma);
14          x = fact * x_new; }
15     return x; }
```

Figure 4.18: A simplification of the CORDIC algorithm

4.18). The only difference is in the choice of the angles: while code 4.2 selects angles that are optimal from a computational point of view, here the angles makes a perfect bisection. We will prove the correctness of the original algorithm with an extended theorem in section 4.4.6.

While the method of section 4.3 is based only on the finite precision semantics, here part of the proof is based on the exact semantics. In particular, this means that the part of the code we extract got a mathematical meaning which has more properties.

The first step of the method consists of considering a possible matching against the template. To do so, we need first to partition variables into the defined abstract variables and then to split the code such that all parts satisfy the conditions about their right to access and to modify variables (4.2). The next step consists of defining the h function which can not be retrieved from the code. Finally, all conditions have to be proven. Here, we only prove the mathematical part, the exact computation of the constants for the finite precision part does not bring anything to our example.

Scheme instantiation

We recall that our pattern is the following:

```
foo(m){
      (n,i) = Init();
      while(! S(m,n,i)) {
```

```
      (i,n) = C(m,n,i);
          m = R(i,m); }
    return m; }
```

First, we need to find out which variables are part of **m**, **n** and **i** and which parts of the code correspond to `C` and `R`.

The variable **m** has to be the input and the output which is not the case for CORDIC code 4.2. However, as we explain in remark 3, we still can merge `beta` and `x` into **m**. Here, we also need to include `y` and `theta` into **m**. Since there are several variables, we have to choose a distance: we take the d_1 distance that is the sum of the distance of each component (definition 2.2.5).

When doing this merge, we have to recall that only `beta` is the actual input: the closeness property does not have to deal with initial perturbations of `x` or `theta` for instance, since they are initialized at the beginning of the program.

Once, we have instantiated **m**, we have to decide where to split the body of the loop to define `C` and `R`. `R` can only modify variables in **m**, but in fact, it can contains local variables: variables that are initialized from variables **m** and **i** at each loop iteration and that are never used outside of `R`. The variable `sigma` is modified inside the `if` branch: it is not a variable of `R`. Hence, since this variable is set in line 9 we can conclude that `C` begins line 5 and ends line 9 while `R` begins line 10 and ends line 14.

```
C(<x,y,beta>,<>,<sigma,n>) {
  if( beta > theta )
    sigma = 1;
  else
    sigma = -1;
  sigma =  sigma * M_PI / (1 << i);
  i = i + 1; //due to the syntactic sugar
  return <sigma,i>; }

R(<sigma,n>,<x,y,beta>) {
 theta = theta + sigma;
 fact = cos(sigma); // Value stored
 atan_sigma = atan(sigma); // Value stored
 x_new = x + y * atan_sigma;
 y = fact * (y - x * atan_sigma);
 x = fact * x_new;
 return <x,y,beta>;}
```

Finally, the variable **i** contains all other variables than the ones in m and which are local variables of R or C: **i** contains i and sigma (there is no variable in **n**) .

Now we have instantiated C and R, we need to express the exact semantics of R in order to define the rewrite system $\mathcal{R}$.

In the exact semantics, we have:

$$[\![\texttt{R(< sigma,i >, < x,y,beta,theta >)}]\!] = (\mathrm{rot}_{\sigma i}(x,y), \beta, \theta + \sigma n)$$

where rot_α is the rotation function of angle α of the vector (x,y).

From $[\![\mathtt{R}]\!](\langle\sigma,i\rangle,\langle x,y,\beta,\theta\rangle)$, we get our rewrite system where the objects are tuples with a point in $\mathbb{R}^4$ ($\langle x,y,\beta,\theta\rangle$) and the rules are the partial functions $\lambda xy\beta\theta.[\![\mathtt{R}]\!](\sigma,i,x,y,\beta,\theta)$ where i and σ are finite. So, the rewrite system that corresponds to R is the set of rotations with an angle α where

$$\alpha \in \{\pm 2^{-i}\pi | i \in [\![1,n]\!]\}$$

We denote by R_α the corresponding rule.

Finally, we have to define the h function on $\mathbb{R}^m$. This function has to measure the progress done by the algorithm while successive iterations are achieved. Since the algorithm stops when β and θ are close, it is natural to define h by:

$$h(x,y,\beta,\theta) = |\beta - \theta|$$

Proofs of the conditions

Now, we prove the conditions of section 4.4.3.

Condition 4.4.1 We have to prove the local confluence of $\overset{>}{\to}$.

Proof. We denote by $m_{(\alpha,\beta)} \in \mathbb{R}^m$, the tuple $(\cos(\alpha),\sin(\alpha),\beta,\alpha)$. When there is no ambiguity about β (which is invariant by the rewrite rules) we write m_α instead of $m_{(\alpha,\beta)}$. We can check that $R_\gamma(m_\alpha) = m_{\gamma+\alpha}$ since the rules are rotation based. The subsystem $\overset{>}{\to} \subset \to$ just allows rules $m_\alpha \to m_{\alpha'}$ where $|\alpha' - \beta| < |\alpha - \beta|$.

Consider the fork:

$$m_{\alpha_1} \overset{<}{\leftarrow} m_\alpha \overset{>}{\to} m_{\alpha_2}$$

We can remark that all the allowed angles are multiple of $2^{-n}\pi$. Consider the value ρ such that

$$\beta - 2^{-n-1}\pi \le \rho \le \beta + 2^{-n-1}\pi$$

and

$$|\rho - \alpha| = k2^{-n}\pi.$$

In words, the closest value of β which is reachable from α in $\rightarrow$.

If there is two values for ρ (limit case), then the proof fails. This is not a problem, however, since such a failure set is enumerable. In other cases, me can reduce

$$m_{\alpha_1} \xrightarrow{>}{}^{*} \rho \; {}^{*} \xleftarrow{<} m_{\alpha_2}$$

by a succession of rules $R_{2^{-n}\pi}$ and $R_{-2^{-n}\pi}$.

□

Condition 4.4.2 We have to prove the termination of the rewriting system $\xrightarrow{>}$.

Proof. From $m_\theta \in \mathbb{R}^m$, the set reachable points by $\xrightarrow{>}$ is the finite set

$$\{\theta + k2^{-n}\pi | k \in \mathbb{Z} \wedge |\theta + k2^{-n} - \beta| \leq |\theta - \beta|\}.$$

Since $\xrightarrow{>}$ is acyclic by construction, $\xrightarrow{>}$ is terminating.

□

Condition 4.4.3 We have to prove:

$$a \rightarrow b \implies \exists c, a \xrightarrow{>}{}^{*} c \; {}^{*} \xleftarrow{<} b.$$

Proof. The $\rightarrow$ system is symmetric. So either $a \xrightarrow{>} b$ and the property holds for $c = b$ or $a \xleftarrow{<} b$ and the property holds for $c = a$. □

Condition 4.4.5 At the end of the computation, in the exact semantics, the final value of m is such that $m \xrightarrow{>}{}^{*} \bar{m}$ implies $d(\bar{m}, m) \leq \varepsilon_s$.

Proof. The modified algorithm of CORDIC does a strict bisection of angles. At the first iteration, the only possible angle for α is $\pi/2$. At the second iteration, the possibles angles reachable by $[\![foo]\!]$ are $\pi/4$ and $3\pi/4$ but not $\pi/2$ anymore. After n iterations, the possibles angles are $(2k+1)\pi/2^n$, $k \in \mathbb{N}$. Since the possible angles of our rewrite system are in $k\pi/2^n$, $k \in \mathbb{N}$, the maximal distance for α is $\pi/2^n$. This distance is actually reached when $\beta = \pi/2$: in that case, the exact value is reached after one iteration of the algorithm but since the algorithm does not stop, the

final value is just close to $\pi/2$. Since we consider the L_1 distance, we have

$$d(\bar{m},m) = \sup_{d(\alpha_1,\alpha_2)\leq\pi/2^n} (\cos\alpha_1 - \cos\alpha_2) + (\sin\alpha_1 - \sin\alpha_2) + |\alpha_1 - \alpha_2|$$

And then

$$\varepsilon_s \leq (1+\sqrt{2})\frac{\pi}{2^n}$$

□

Condition 4.4.4 The function $[\![foo]\!]$ is $P(k_e,\varepsilon_e)$.

Proof. From the previous proof, we know that the final values for α are in $(2k+1)\pi/2^n$, $k \in \mathbb{N}$. We also know that the result of $[\![foo]\!]$ is the tuple $m_\alpha = (\cos\alpha, \sin\alpha, \alpha, \beta)$ with $|\alpha - \beta| \leq \pi/2^n$. By definition, the $P(k_e,\varepsilon_e)$ property provide a bound on the following value.

$$[\![foo]\!](m_{0,\beta}, m_{0,\beta'}) = d((\cos\alpha, \sin\alpha, \alpha, \beta-\alpha), (\cos\alpha', \sin\alpha', \alpha', \beta'-\alpha'))$$

With the last inequality, we get

$$|\alpha - \alpha'| \leq \frac{\pi}{2^{n-1}} + |\beta - \beta'|$$

By use of trigonometric property, we get:

$$[\![foo]\!](m_{0,\beta}, m_{0,\beta'}) = \sqrt{2}d(m_{0,\beta}, m_{0,\beta'}) + \sqrt{2}\frac{\pi}{2^{n-1}}$$

□

Condition 4.4.6 This condition expresses that, if the control flows were the same, the finite-precision semantics and the exact semantics would be close to each other.

Proof. The program does exactly n iterations of the loop. All the operations in `R` are basic k-Lipschitz arithmetic operations in a bounded domain. All these instructions should enjoy the closeness property in a safe finite-precision arithmetic representation. Since closeness is, somehow, compositional (proposition 2.3.4), the two control flows are close to each other.

The computation of the actual constant is too dependent on the architecture to be computed here. □

Condition 4.4.7 In the finite-precision semantics, when the stopping condition is reached, the final value m' is such that

$$\exists z' \in \bar{\mathbb{R}^m} d(m', z') \leq \varepsilon'_s$$

Proof. The z' we have to exhibit does not have to have any relationship with the z of condition 4.4.5. Moreover, any m_α with

$$|\alpha - \beta| \leq 2^{-n-1}\pi$$

is in a normal form. So, to obtain ε'_s, we just have to compute the maximal distance that can exists between the α' and β' of m' and then subtract $2^{-n-1}\pi$. To find this value, we need to analyze the code of `C` in the finite semantics. Here, again, we do not do this analyze, but since the control flow depends on `theta` and always selects the branch that reduces `theta` the most it should not be a source of difficulty. □

4.4.5 Application to Dijkstra's algorithm

In this section, we apply our method to Dijkstra's shortest path algorithm (described in 4.1.2). We prove the closeness property according to the d_1 distance for the input (the list of weighted edge) and for the output (the list of the minimal length for each node).

Scheme instantiation

Here, **m** also contains the input and the output, so **m** is the tuple `<graph, pathestimate>`. The distance is then the d_1 metric on the pair.

Then we need to split the body of the loop into code `C` and code `R`. This is not possible from a syntactic point of view since the main loop contains another loop. However, the code 4.7 can be rewritten in the equivalent code 4.19.

With this new code, the tuple **i** is instantiated by `<u,i>` an oriented edge: the node i is the node to potentially update from the value of the node u and the length of the edge `graph[u][i]`. The tuple **n** contains the auxiliary variables `<count, mark>` used to mark the nodes which already have their definitive value in `pathestimate`.

The part that chooses the rules is the following.

```
C (<graph, pathestimate>, <count, mark>, <u, i>) {
    if(i == n){
      u = minimum(pathestimate, mark, n);
      mark[u] = 1;
      count = count + 1;
      i = 0; }
```

```
int[] dijkstra( int graph[n][n]){
   int pathestimate[n],mark[n];
   int source,i,j,u,predecessor[n],count=0;
   int minimum(int a[],int m[],int k);
   for(j=1;j<=n;j++){
      mark[j]=0;
      pathestimate[j]=999;
      predecessor[j]=0;}
   source=0;
   pathestimate[source]=0;
   while(count<n){
     if(i==n){
       u=minimum(pathestimate,mark,n);
       mark[u]=1;
       count=count+1;
       i=0;}
     else {
          i=i+1;}
     if(pathestimate[i]>pathestimate[u]+graph[u][i]){
          pathestimate[i]=pathestimate[u]+graph[u][i];
          predecessor[i]=u;}}
     return pathestimate;}

int minimum(int a[],int m[],int k){
     int mi=999;
     int i,t;
     for(i=1;i<=k;i++){
          if(m[i]!=1){
               if(mi>=a[i]){
                    mi=a[i];
                    t=i;}}}
     return t;}
```

Figure 4.19: A rearrangement of the Dijktra's code

```
      else {
           i = i + 1; }
      return <u, i>;
}
```

The rewrite rules are implemented by:

```
R (<u, i>, <graph, pathestimate>) {
        if(pathestimate[i] > pathestimate[u] + graph[u][i]){
             pathestimate[i] = pathestimate[u] + graph[u][i];
        }
    return <pathestimate,graph>;
}
```

For readability reason, we denote by $\mathcal{G}$ the value of `graph` in the exact semantics and $\mathcal{P}$ the value of `pathestimate` in the exact semantics. So $\mathcal{G} = [\![\texttt{graph}]\!]$ and $\mathcal{P} = [\![\texttt{pathestimate}]\!]$. In addition, we denote by $\mathcal{P}_u$ the value in `pathestimate[u]`.

With this notation, the exact semantics of `R` is the function that returns the identity excepts for `pathestimate[i]`:

$$[\![\texttt{R(<u,i>,<graph,pathestimate>)}]\!]_i = \min(\mathcal{P}_i, \mathcal{P}_u + \mathcal{G}_{u,i})$$

So our rewrite system $\rightarrow$ contains rules indexed by two nodes u and i. Finally, since the program is computing a minimal possible value for $\mathcal{P}$, it is natural to define h as the sum of the values of all nodes of pathestimate. Hence, if the minimal value is reached, then h cannot decrease anymore which means $\mathcal{P}$ is in a normal form. In that setting, we have $\stackrel{>}{\rightarrow} = \rightarrow$ (strictly speaking the identity rules are removed).

Proofs of the conditions

We have to prove that the conditions hold for the given instantiations. Once again, we do not compute the constants related to the finite precision semantics.

Condition 4.4.1 Local confluence.

Proof. Let $a \in \mathbb{R}^m$ and

$$c \xleftarrow[<u',v'>]{} a \xrightarrow[<u,v>]{} b$$

two pairs of nodes where we apply the rules. We can compute that

$$c \xrightarrow[<u,v>]{\geq} e \xrightarrow[<u',v'>]{\geq} f \xleftarrow[<u,v>]{\leq} d \xleftarrow[<u',v'>]{\leq} b$$

where $x \xrightarrow[<u,v>]{\geq} y$ means that when we apply the rules that updates v from u and the length of the edge (u,v) either $x \xrightarrow{>} y$ (the value $\mathcal{P}_y$ has been updated with a smaller value) or $x = y$ and (the rules does not do anything). □

Condition 4.4.2 Termination of $\xrightarrow{>}$.

Proof. For each node, the possible values for $\mathcal{P}_u$ are in

$$\{\sum_{i,j\in S} \mathcal{G}_{i,j} | S \subset [\![1,n]\!]^2\}.$$

This is due to the fact a new value is computed from already computed values that are already a sum of edge length. Then, since all values are positive, summing twice an edge always increases the estimated value. Hence, it is never less than a previous value. So since values belongs to a finite set and there is no circle, the system terminates. □

Condition 4.4.3

$$\forall a,b \in \mathbb{R}^m, a \to c \implies \exists c \in \mathbb{R}^m a \xrightarrow{>}{}^* c \,{}^* \xleftarrow{<} b$$

Proof. Since $\to \; = \xrightarrow{>}$, this property is immediate. □

Condition 4.4.4 The function $[\![foo]\!]$ is $P(k_e, \varepsilon_e)$.

Proof. The function $[\![foo]\!]$ is the Dijkstra's algorithm of 4.1.2 which is an n-Lipschitz function. So the function $[\![foo]\!]$ is hence $P(n,0)$. □

Condition 4.4.5 At the end of the computation, in the exact semantics, the final value of m is such that $m \xrightarrow{>}{}^* \bar{m}$ implies $d(\bar{m}, m) \leq \varepsilon_s$.

Proof. Unlike for CORDIC, Dijkstra's algorithm computes exactly the minimum length for each node. Since h is defined as the sum for all node, $h([\![foo]\!](\mathcal{G}))$ is the actual minimal value. In particular, we have $\varepsilon_s = 0$. □

Condition 4.4.6 The closeness property holds between $[\![foo]\!]'$ and p (from definition 4.4.10).

Proof. By looking at `C`, we can see that the number of rules that can be applied is bounded by n^2. More precisely `i` and `n` are initialized to 0. Then either `i` increase by 1 or `count` increase by 1 and `i` is reset to 0. Finally `count` has to be less than n. □

Condition 4.4.7 In the finite-precision semantics, when the stopping condition is reached, the final value m' is such that

$$\exists z' \in \bar{\mathbb{R}^m} d(m', z') \leq \varepsilon'_s$$

Proof. Adding a positive value never renders a smaller result in floating or fixed-point representation. Moreover, looking for the minimum cannot produce errors (by returning a number which is not the minimal). To prove this condition, we need to ensure these assertions are true (formally a finite precision representation might not provide these guarantees even if the floating and fix-point representation do). We also notice that reading the value of an edge does not change its value (otherwise the algorithm would not be safe).

Hence, the only sources of error are the additions. This means that, while updating an edge (u,v), we get

$$|\mathcal{P}'_u - \min_v \mathcal{P}'_v - \mathcal{G}_{(u,v)}| \leq \varepsilon \tag{4.13}$$

We have to find a variable $z' \in \bar{\mathbb{R}^m}$, in our case, this means exhibiting a pair $(\mathcal{P}^z, \mathcal{G}^z)$. We take, $\mathcal{P}^z = \mathcal{P}'$ and $\mathcal{G}^z$ such that in case an edge (u,v) has actually been used to update a node, we define $\mathcal{G}^z_{u,v} = \mathcal{P}'_u - \mathcal{P}'_v$ and $\mathcal{G}^z_{u,v} = \mathcal{G}_{u,v}$ otherwise. This pair is terminating by construction and we derive from equation 4.13 and the fact there are n nodes:

$$d(\mathcal{G}^z, \mathcal{G}) = \sum_{u,v} |\mathcal{G}^z_{u,v} - \mathcal{G}_{u,v}| \leq n^2 \varepsilon$$

□

4.4.6 Approximate confluence

As we have seen for the CORDIC algorithm, sometimes programs do not achieve strict confluence but just an "approximate" one. In that case, our theorem does not apply. To deal with this case, we propose here an extension to our theorem (i.e. a more general theorem but with a harder to prove condition).

First, we define formally what is "approximate" confluence.

Definition 4.4.13 (Approximate confluence)**.** *A rewrite system $\rightarrow$ is approximately confluent when for any $\varepsilon \in \mathbb{R}^+$, there exists e and f such that $d(e,f) \leq \varepsilon$ and:*

$$b \,{}^*\!\leftarrow a \rightarrow^* c \implies b \rightarrow^* e \wedge f \,{}^*\!\leftarrow c$$

This definition is interesting only for non terminating rewrite system since otherwise this definition is equivalent to global confluence.

Proposition 4.4.6. *The two propositions are equivalent:*

- *A rewrite system is globally confluent*
- *A rewrite system $\rightarrow$ is approximately confluent and terminating*

Proof. A globally confluent rewrite system implies approximate confluence in any case (with $d = e$).

For the other direction, let a_1 and a_2 to be two normal forms of a. The approximate confluence applied on $a_1 \,{}^*\!\leftarrow a \rightarrow^* a_2$ states $d(a_1,a_2) \leq \varepsilon$ for all ε. So we conclude $a_1 = a_2$. □

With this definition, however, there is no equivalent to the Newman's lemma that allows us to prove only local confluence. Then, even if we assume the approximate confluence for $\xrightarrow{>}$, proposition 4.4.5 would not hold. Here, since we are just fixing the CORDIC example with the original code, we do not try to find convenient conditions, we just assume a condition to replace proposition 4.4.5.

Condition 4.4.8. *The following assertion holds.*

$$\forall x,a,b \in \mathbb{R}^m, \forall \varepsilon \in \mathbb{R}^+, a \,{}^*\!\leftarrow x \rightarrow^* b \implies \exists e,f, a \xrightarrow{>}{}^* e \wedge f \,{}^*\!\xleftarrow{<} b \wedge d(e,f) \leq \varepsilon$$

Since condition 4.4.1, condition 4.4.2 and condition 4.4.3 was mainly used to prove proposition 4.4.5, they are not useful now and they are not part of our new theorem.

On the other hand, we have to change our condition 4.4.7, since the set $\bar{\mathbb{R}^m}$ is not defined anymore. We replace it by this one.

Condition 4.4.9. *In the finite-precision semantics, when the stopping condition is reached, the final value m' is such that*

$$\exists z', d(m',z') \leq \varepsilon'_s \wedge z' \xrightarrow{>}{}^* z'_f \implies d(z',z'_f) \leq \varepsilon_s$$

With this change, we obtain a new theorem with almost the same conclusion (there is just an additional ε_s) but with less restrictive premisses.

Theorem 4.4.2. *If conditions 4.4.8, 4.4.4 4.4.7, 4.4.6 and 4.4.7 hold, the function $[\![foo]\!]'$ computed in a finite-precision semantics and the function $[\![foo]\!]$ that would be computed in the idealized semantics are (k,ε)-close with $k = k_e$ and $\varepsilon = k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 3\varepsilon_s + \varepsilon'_s$.*

$$\forall x, y \in \mathbb{R}^m, d([\![foo]\!](x), [\![foo]\!]'(y)) \leq k_e d(x,y) + k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 3\varepsilon_s + \varepsilon'_s$$

The proof scheme of this new theorem is basically the same as the one of theorem 4.4.1. But, since we have weaker hypothesis, the details of the proof changes and there is no exactly identical part in the proof.

Proof. Let $x, y \in \mathbb{R}^m$ and $\varepsilon_0 \in \mathbb{R}^+$, we have to prove there exist k and ε such that

$$d([\![foo]\!](y), [\![foo]\!]'(x)) \leq kd(y,x) + \varepsilon + \varepsilon_0.$$

From a triangular inequality, we get

$$d([\![foo]\!](y), [\![foo]\!]'(x)) \leq d([\![foo]\!](y), [\![foo]\!](x)) + d([\![foo]\!](x), [\![foo]\!]'(x))$$

From condition 4.4.4, we have

$$d([\![foo]\!](y), [\![foo]\!](x)) \leq k_e d(x,y) + \varepsilon_e. \tag{4.14}$$

So, now, we have to get a bound for $d([\![foo]\!](x), [\![foo]\!]'(x))$.

Now, Condition 4.4.9 provides some z' with

$$d([\![foo]\!]'(x), z') \leq \varepsilon'_s \tag{4.15}$$

and

$$z' \xrightarrow{\geq}{}^* z'_f \implies d(z', z'_f) \leq \varepsilon_s. \tag{4.16}$$

From inequality 4.15 and the inequality of the Condition 4.4.6, we derived with a triangular inequality: $d(z', p(x)) \leq k_f d(x,x) + \varepsilon_f + \varepsilon'_s$. And then:

$$d(z', p(x)) \leq \varepsilon_f + \varepsilon_s. \tag{4.17}$$

Now, from condition 4.4.8, we have, since $z' \to^* [\![foo]\!](z')$:

$$\exists e, f, [\![foo]\!](z') \xrightarrow{\geq}{}^* e \wedge f \; {}^* \xleftarrow{\leq} z' \wedge d(e,f) \leq \varepsilon_0.$$

Then from equation 4.16 $d(z', [\![foo]\!](z')) \leq \varepsilon_s$. From condition 4.4.5, we have:

$$d([\![foo]\!](z'), e) \leq \varepsilon_s.$$

Finally, we derive from a triangular inequality with these three last inequalities

$$d([\![foo]\!](z'), z') \leq 2\varepsilon_s + \varepsilon_0. \tag{4.18}$$

By definition of the path d, we have $x \to^* p(x)$. Since $x \to^* p(x)$ and $x \to^* [\![foo]\!](x)$, we derived from condition 4.4.8:

$$\exists u, v, p(x) \xrightarrow{\geq}{}^* u \wedge v \,{}^* \xleftarrow{\leq} [\![foo]\!](x) \wedge d(u, v) \leq \varepsilon_0.$$

Then with condition 4.4.5, we derive:

$$d([\![foo]\!](p(x)), v) \leq \varepsilon_s. \tag{4.19}$$

Condition 4.4.4 on $p(x)$ and z':

$$d([\![foo]\!](p(x)), [\![foo]\!](z')) \leq k_e d(p(x), z') + \varepsilon_e.$$

By using equation 4.17, we get:

$$d([\![foo]\!](p(x)), [\![foo]\!](z')) \leq k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e.$$

Then, a triangular inequality with equation 4.19

$$d(v, [\![foo]\!](z')) \leq k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + \varepsilon_s.$$

Then, a triangular inequality with equation 4.18

$$d(v, z') \leq k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 2\varepsilon_s + \varepsilon_0.$$

Then condition 4.4.5 on $[\![foo]\!](x)$ and the fact that $d(u, v) \leq \varepsilon_0$, allow us to derive with a triangular inequality:

$$d([\![foo]\!](x), z') \leq k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 3\varepsilon_s + 2\varepsilon_0. \tag{4.20}$$

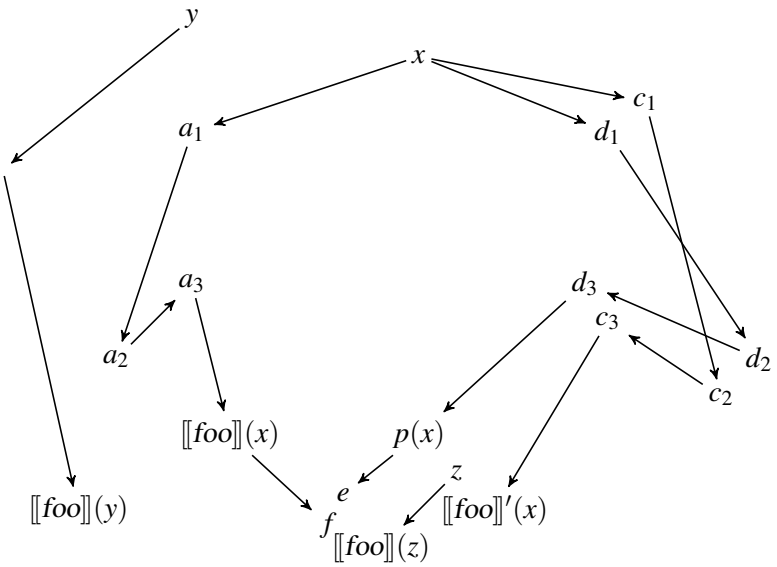

Figure 4.20: A picture of the proof

Now, we do a triangular inequality from inequalities 4.20, 4.14 and 4.15:

$$d([\![foo]\!](y),[\![foo]\!]'(x)) \leq k_e d(x,y) + \varepsilon_e + k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 3\varepsilon_s + 2\varepsilon_0 + \varepsilon'_s.$$

Since this result is true for all $\varepsilon_0 \in \mathbb{R}^+$, we conclude $[\![foo]\!]$ and $[\![foo]\!]'$ are $(k_e, k_e(\varepsilon_f + \varepsilon_s) + \varepsilon_e + 3\varepsilon_s + \varepsilon'_s)$-close. □

Figure 4.20 represents the paths and elements used in the proof. As we can see, the scheme of the proof is almost identical of figure 4.17.

Application to CORDIC Since the analysis is almost the same than in subsection 4.4.4, we just mention here the differences. Here, the rewrite system is now composed by all rotation rules R_α with $\alpha \in \mathbb{R}$ and not only $\alpha \in \{\pm 2^{-i}\pi | i \in [\![1,n]\!]\}$ as in subsection 4.4.4. Now, we have to prove again all conditions for the program 4.2 with a new rewrite system. Condition 4.4.8: the following assertion holds.

$$\forall x,a,b \in \mathbb{R}^m, \forall \varepsilon \in \mathbb{R}^+, a \,{}^*\!\leftarrow x \rightarrow^* b \implies \exists e,f, a \xrightarrow{>}{}^* e \wedge f \,{}^*\!\xleftarrow{<} b \wedge d(e,f) \leq \varepsilon$$

Proof. Let $m_{\gamma,\beta} \,{}^*\!\xleftarrow{<} m_{\alpha,\beta} \xrightarrow{>}{}^* m_{\theta,\beta}$, we have

$$m_{\gamma,\beta} \xrightarrow{>} m_{\beta,\beta} \xleftarrow{<} m_{\theta,\beta}$$

So, we can choose $m_{\beta,\beta} = e = f$ in the condition. □

Condition 4.4.5:

Proof. In the exact semantics, after the n iterations we have $|\alpha - \beta| \leq \delta$ (proposition 4.1.2) Since $m_x \xrightarrow{>}{}^* m_y$ implies $d(x,\beta) > d(y,\beta)$, we can conclude that $d(x,y) \leq 2\delta$. Finally, we have

$$\varepsilon_s = \sup_{\beta \in [0,\pi/2]} 2\delta + |\cos(\beta - \delta) - \cos(\beta + \delta)| + |\cos(\beta - \delta) - \cos(\beta + \delta)|$$

With trigonometric properties, we can get

$$\varepsilon_s = 2\delta + 2\sqrt{2}\delta$$

□

Condition 4.4.4 states that the program computes a function which is $P(k,e)$ in the exact semantics.

Proof. The proof is the same as the one of section 4.4.4 except that $\pi/2^{n-1}$ is replaced by $\tan^{-1}(2^{n-1})$ according to proposition 4.1.2. □

Condition 4.4.6 concerns the closeness of the program when the control flow is respected.

Proof. There is no change in the proof of section 4.4.4. □

Condition 4.4.9 states that in the finite-precision semantics, when the stopping condition is reached, the final value m' is such that

$$\exists z',\ d(m',z') \leq \varepsilon'_s \wedge z' \xrightarrow{>}{}^* z'_f \implies d(z',z'_f) \leq \varepsilon_s$$

Proof. We look for an element $m_(\rho,\rho)$ to be z'. Indeed, such an element is in a normal form. Then, since as for the initial proof from section 4.4.4, we still can prove that the final value of α and β is bounded by some $\tan^{-1}(2^{n-1}) + \varepsilon$. Then, it is possible to compute a bound for ε'_s. □

4.5 Conclusion

In this chapter, we have investigated non compositional techniques for analysis.

In our first try, our goal was to analyze the robustness of the code directly in its finite-precision semantics. The idea was to identify a pattern that decomposes the code into smaller parts and then to analyze these parts and some relationship between them. The pattern we

investigate provides an easier analysis of the parts of code. However, our main problem was to prove condition 4.3.2 that states a relationship between the two main parts of the template. Since, we only study the finite precision semantics, relationships are harder to prove because they cannot be based on a mathematical statement. In fact, we show that this technique works, but, in both studied examples, it is difficult to prove condition 4.3.2.

From this first try, we conclude that proving robustness from scratch was a too difficult goal. In addition, we also assume that, when an algorithm is not locally robust, its proof of correctness is not obvious. So, there are good reasons to believe that it has been proved mathematically (in its exact semantics) before to be written. Therefore, we decide, in our second attempt, to mix conditions about the exact semantics with conditions about the finite semantics. The result we got is much more convincing than the first try. First, the proved property is stronger since it is not just the $P(k,\varepsilon)$ property that has been proved but the closeness to the exact semantics. Next, the conditions to prove are easier and more natural than in the direct analysis, essentially because the difficulty is now concentrated on the correctness of the algorithm in its exact semantics.

As a future work, it might be interesting to investigate close problems to enlarge the scope of this method. For instance, a difficult problem, which has not been studied here, is the question of the stopping condition of loops. Indeed, on some algorithms (like computation of series), the number of loop iterations that has to be done is not the same in the finite precision and in the exact precision. In some cases, the program in the finite precision diverges while it converges in the exact semantics. Providing simple conditions to check how the finite precision semantics differs from the exact algorithm would be a nice extension of this work.

$\star$

CHAPTER 5

CONCLUSION

In this thesis, we have investigated the problem of how to extend a proof of an algorithm in the exact semantics into a proof about its implementation in finite precision. In the first chapter, we details the behavior of finite precision representations and we briefly presented standard technique to analyze code. From these description we provide general definitions: the $P(k,\varepsilon)$-property and the (k,ε)-closeness property that are easy to use in a manual proof and which can be stated by most static analyzers since these definition are not optimized. Then from this general framework, we have studied the problem of implementation of differential privacy and the problem of confluent programs that cannot be analyzed by compositionality.

The question of leakage in differential privacy are well studied while the question of its implementation is a new problematic [Mir12]. In general, implementation questions are considered as a special problem which is only in the domain of program analysis. This follows the idea that, once an algorithm is proved mathematically, the question of its implementation is just a technical question. By providing a mathematical statement about how the differential privacy is altered by errors and by proving that there is no implementation in finite precision that can satisfy the designed algorithm 1, we stress that implementation should also be seen as a main concern. Although our theorem, by considering a simple characterization of errors, may not be optimal in its constants, it allows to bridge the gap between results that are obtained by software and not directly related mathematical statements. In particular, when we applied the theorem in the standard case, we found that the result was not convincing. Then we was able to provide a new mechanism that is more efficient than the one proposed in [Mir12].

In the chapter about global behavior, we address the problem of changes of the control flow induced by rounding errors. When, the two branches of the control flow do not share any similarity, it is not possible to consider a property about their union. A standard compositional analysis is then not possible. We start our investigation from this fact and try to propose another kind of

analysis. The framework we have developed considers control flow errors as non determinism and provides robustness through a kind of confluence property. The theorem we have provided bridge the gap between the algorithm proof and the automated analysis by mixing conditions on the exact semantics and the finite semantics. We do not know how large is the class of algorithms that can be analyzed this way, but our main goal was to open research directions about interleaving between exact and finite precision semantics.

BIBLIOGRAPHY

[ABCP13] Miguel Andrés, Nicolás Bordenabe, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Geo-indistinguishability: Differential privacy for location-based systems. In *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, 2013. 28, 30, 69

[ACPS12] Mário S. Alvim, Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Geoffrey Smith. Measuring information leakage using generalized gain functions. In *Proceedings of the 25th IEEE Computer Security Foundations Symposium (CSF)*, pages 265–279, 2012. 73

[BCP09] Christelle Braun, Konstantinos Chatzikokolakis, and Catuscia Palamidessi. Quantitative notions of leakage for one-try attacks. In *Proceedings of the 25th Conf. on Mathematical Foundations of Programming Semantics*, volume 249 of *Electronic Notes in Theoretical Computer Science*, pages 75–91. Elsevier B.V., 2009. 73

[BF07] Sylvie Boldo and Jean-Christophe Filliatre. Formal verification of floating-point programs. In Peter Kornerup and Jean-Michel Muller, editors, *Proceedings of the 18th IEEE Symposium on Computer Arithmetic, June 25–27, 2007, Montpellier, France*, pages 187–194, pub-IEEE:adr, 2007. IEEE Computer Society Press. 18, 19

[BKOB12] Gilles Barthe, Boris Köpf, Federico Olmedo, and Santiago Zanella Béguelin. Probabilistic relational reasoning for differential privacy. In *Proceedings of the 39th Annual ACM Symposium on Principles of Programming Languages (POPL)*. ACM, 2012. 28

[BP05] Mohit Bhargava and Catuscia Palamidessi. Probabilistic anonymity. In Martín Abadi and Luca de Alfaro, editors, *Proceedings of the 16th International Conference on Concurrency Theory (CONCUR 2005)*, volume 3653 of *Lecture Notes in Computer Science*, pages 171–185. Springer, 2005. 73

[BP12] Michele Boreale and Francesca Pampaloni. Quantitative multirun security under active adversaries. In *Proceedings of the Nineth International Conference on Quantitative Evaluation of SysTems (QEST)*, pages 158–167. IEEE Computer Society, 2012. 73

[BPP11] Michele Boreale, Francesca Pampaloni, and Michela Paolini. Asymptotic information leakage under one-try attacks. In Martin Hofmann, editor, *Proceedings of the 14th International Conference on the Foundations of Software Science and Computational Structures (FOSSACS'11)*, volume 6604 of *Lecture Notes in Computer Science*, pages 396–410. Springer, 2011. 73

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of the 4th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, POPL '77, pages 238–252, New York, NY, USA, 1977. ACM. 13

[CDJ08] T. Champion, L. De Pascale, and P. Juutinen. The ∞-wasserstein distance: Local solutions and existence of optimal transport maps. *SIAM Journal on Mathematical Analysis*, 40(1):1–20, 2008. 54

[CGH+96] R. M. Corless, G. H. Gonnet, D. E. G. Hare, D. J. Jeffrey, and D. E. Knuth. On the lambert w function. In *ADVANCES IN COMPUTATIONAL MATHEMATICS*, pages 329–359, 1996. 70

[CGL10] Swarat Chaudhuri, Sumit Gulwani, and Roberto Lublinerman. Continuity analysis of programs. In Manuel V. Hermenegildo and Jens Palsberg, editors, *POPL*, pages 57–70. ACM, 2010. 14, 85

[CGLN11] Swarat Chaudhuri, Sumit Gulwani, Roberto Lublinerman, and Sara NavidPour. Proving programs robust. In Tibor Gyimóthy and Andreas Zeller, editors, *SIGSOFT/FSE'11 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering (FSE-19) and ESEC'11: 13rd European Software Engineering Conference (ESEC-13), Szeged, Hungary, September 5-9, 2011*, pages 102–112. ACM, 2011. 14, 31, 42

[CHM05] David Clark, Sebastian Hunt, and Pasquale Malacaria. Quantified interference for a while language. In *Proceedings of the Second Workshop on Quantitative Aspects of Programming Languages (QAPL 2004)*, volume 112 of *Electronic Notes in Theoretical Computer Science*, pages 149–166. Elsevier Science B.V., 2005. 73

[CPP08] Konstantinos Chatzikokolakis, Catuscia Palamidessi, and Prakash Panangaden. Anonymity protocols as noisy channels. *Information and Computation*, 206(2–4):378–401, 2008. 73

[Dij71] Edsger W. Dijkstra. A short introduction to the art of programming. circulated privately, August 1971. 82, 83

[DLAMV12] Yevgeniy Dodis, Adriana López-Alt, Ilya Mironov, and Salil P. Vadhan. Differential privacy with imperfect randomness. In Reihaneh Safavi-Naini and Ran Canetti, editors, *Advances in Cryptology—CRYPTO 2012*, volume 7417 of *Lecture Notes in Computer Science*, pages 497–516. Springer, 2012. 46

[DMNS06] Cynthia Dwork, Frank Mcsherry, Kobbi Nissim, and Adam Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin, editors, *In Proceedings of the Third Theory of Cryptography Conference (TCC)*, volume 3876 of *Lecture Notes in Computer Science*, pages 265–284. Springer, 2006. 28

[Dwo06] Cynthia Dwork. Differential privacy. In Michele Bugliesi, Bart Preneel, Vladimiro Sassone, and Ingo Wegener, editors, *33rd International Colloquium on Automata, Languages and Programming (ICALP)*, volume 4052 of *Lecture Notes in Computer Science*, pages 1–12. Springer, 2006. 28

[Dwo11] Cynthia Dwork. A firm foundation for private data analysis. *Communications of the ACM*, 54(1):86–96, 2011. 39, 40

[EKL06] T. Eltoft, Taesu Kim, and Te-Won Lee. On the multivariate Laplace distribution. *IEEE Signal Processing Letters*, 13(5):300–303, May 2006. 34

[GHH+13] Marco Gaboardi, Andreas Haeberlen, Justin Hsu, Arjun Ravi Narayan, and Benjamin C. Pierce. Linear dependent types for differential privacy. In *Proceedings of the 40th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages (POPL)*, pages 357–370, 2013. 28

[GMP12a] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. A non-local method for robustness analysis of floating point programs. In Herbert Wiklicky and Mieke Massink, editors, *Proceedings 10th Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, volume 85 of *Electronic Proceedings in Theoretical Computer Science*, pages 63–76. Open Publishing Association, 2012. 4

[GMP12b] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. Non-local robustness analysis via rewriting techniques. Available at `http://www.lix.polytechnique.fr/Labo/Dale.Miller/papers/qfm2012.pdf`, 2012. 4

[GMP13] Ivan Gazeau, Dale Miller, and Catuscia Palamidessi. Preserving differential privacy under finite-precision semantics. In Luca Bortolussi and Herbert Wiklicky, editors, *Proceedings 11th International Workshop on Quantitative Aspects of Programming Languages and Systems (QAPL)*, volume 117 of *Electronic Proceedings in Theoretical Computer Science*, pages 1–18. Open Publishing Association, 2013. 4

[Gol91] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Computing Surveys*, 23(1):5–47, March 1991. 11

[Gou01] Eric Goubault. Static analyses of the precision of floating-point operations. In Patrick Cousot, editor, *Static Analysis, 8th International Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer Verlag, 2001. 75

[GP11] Eric Goubault and Sylvie Putot. Static analysis of finite precision computations. In Ranjit Jhala and David A. Schmidt, editors, *Verification, Model Checking, and Abstract Interpretation - 12th International Conference, VMCAI 2011, Austin, TX, USA, January 23-25, 2011. Proceedings*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011. 18

[HD93] Gerben J. Hekstra and Ed F.A. Deprettere. Floating point cordic, 1993. 65

[HO05] Joseph Y. Halpern and Kevin R. O'Neill. Anonymity and information hiding in multiagent systems. *Journal of Computer Security*, 13(3):483–512, 2005. 73

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969. 13

[HR11] Shen-Shyang Ho and Shuhua Ruan. Differential privacy for location pattern mining. In *Proceedings of the 4th ACM SIGSPATIAL International Workshop on Security and Privacy in GIS and LBS (SPRINGL)*, pages 17–24. ACM, 2011. 28

[IEE08] IEEE Task P754. *IEEE 754-2008, Standard for Floating-Point Arithmetic*. IEEE, pub-IEEE-STD:adr, August 2008. 8, 29, 65

[KC93] K. Kota and J. R. Cavallaro. Numerical accuracy and hardware tradeoffs for cordic arithmetic for special-purpose processors. *IEEE Trans. Comput.*, 42(7):769–779, July 1993. 65

[LL07] Ninghui Li and Tiancheng Li. t-closeness: Privacy beyond k-anonymity and l-diversity. In *In Proc. of IEEE 23rd Int'l Conf. on Data Engineering (ICDE'07)*, 2007. 37

[Mal07] Pasquale Malacaria. Assessing security threats of looping constructs. In Martin Hofmann and Matthias Felleisen, editors, *Proceedings of the 34th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL 2007)*, pages 225–235. ACM, 2007. 73

[Mir12] Ilya Mironov. On significance of the least significant bits for differential privacy. In Ting Yu, George Danezis, and Virgil D. Gligor, editors, *Proceedings of the ACM Conference on Computer and Communications Security (CCS)*, pages 650–661. ACM, 2012. 3, 29, 30, 43, 67, 129

[MKA+08] Ashwin Machanavajjhala, Daniel Kifer, John M. Abowd, Johannes Gehrke, and Lars Vilhuber. Privacy: Theory meets practice on the map. In Gustavo Alonso, José A. Blakeley, and Arbee L. P. Chen, editors, *Proceedings of the 24th International Conference on Data Engineering, ICDE 2008, April 7-12, 2008, Cancún, México*, pages 277–286. IEEE, 2008. 28

[MKGV07] Ashwin Machanavajjhala, Daniel Kifer, Johannes Gehrke, and Muthuramakrishnan Venkitasubramaniam. L-diversity: Privacy beyond k-anonymity. *ACM Trans. Knowl. Discov. Data*, 1(1), March 2007. 37

[MS09] Rupak Majumdar and Indranil Saha. Symbolic robustness analysis. In Theodore P. Baker, editor, *IEEE Real-Time Systems Symposium*, pages 355–363. IEEE Computer Society, 2009. 16

[MSW10] Rupak Majumdar, Indranil Saha, and Zilong Wang. Systematic testing for control applications. In *MEMOCODE*, pages 1–10, 2010. 16

[New42] M. H. A. Newman. On Theories with a Combinatorial Definition of Equivalence. *Annals of Mathematics*, 43(2):223–243, 1942. 105

[NS09] Arvind Narayanan and Vitaly Shmatikov. De-anonymizing social networks. In *Proceedings of the 30th IEEE Symposium on Security and Privacy*, pages 173–187. IEEE Computer Society, 2009. 28, 37

[PHW05] Alessandra Di Pierro, Chris Hankin, and Herbert Wiklicky. Measuring the confinement of probabilistic systems. *Theoretical Computer Science*, 340(1):3–56, 2005. 73

[RMFDF08] David Rebollo-Monedero, Jordi Forné, and Josep Domingo-Ferrer. From t-closeness to pram and noise addition via information theory. In Josep Domingo-Ferrer and Yücel Saygin, editors, *Privacy in Statistical Databases*, volume 5262 of *Lecture Notes in Computer Science*, pages 100–112. Springer Berlin Heidelberg, 2008. 37

[Rud86] Walter Rudin. *Real and Complex Analysis*. McGraw-Hill, 3rd edition, 1986. 17

[Smi09] Geoffrey Smith. On the foundations of quantitative information flow. In Luca de Alfaro, editor, *Proceedings of the 12th International Conference on Foundations of Software Science and Computation Structures (FOSSACS 2009)*, volume 5504 of *LNCS*, pages 288–302, York, UK, 2009. Springer. 73

[Swe02] Latanya Sweeney. k-anonymity: A model for protecting privacy. *International Journal of Uncertainty, Fuzziness and Knowledge-Based Systems*, 10(5):557–570, 2002. 36

[Vol59] Jack E. Volder. The CORDIC trigonometric computing technique. *IRE Transactions on Electronic Computers*, EC-8:330–334, 1959. 77, 80

Résumé

Dans cette thèse, nous analysons les problèmes liés à la représentation finie des nombres réels et nous contrôlons la déviation induite par cette approximation. Nous nous intéressons particulièrement à deux problèmes.
Le premier est l'étude de l'influence de la représentation finie sur les protocoles de confidentialité différentielle. Nous présentons une méthode pour étudier les perturbations d'une distribution de probabilité causées par la représentation finie des nombres. Nous montrons qu'une implémentation directe des protocoles théoriques pour garantir la confidentialité différentielle n'est pas fiable, tandis qu'après l'ajout de correctifs, la propriété est conservée en précision finie avec une faible perte de confidentialité.
Notre deuxième contribution est une méthode pour étudier les programmes qui ne peuvent pas être analysés par composition à cause de branchements conditionnels au comportement trop erratique. Cette méthode, basée sur la théorie des systèmes de réécriture, permet de partir de la preuve de l'algorithme en précision exacte pour fournir la preuve que le programme en précision finie ne déviera pas trop.

Abstract

In this thesis, we analyze the problem of the finite representation of real numbers and we control the deviation due to this approximation. We particularly focus on two complex problems.
First, we study how finite precision interacts with differentially private protocols. We present a methodology to study the perturbations on the probabilistic distribution induced by finite representation. Then we show that a direct implementation of differential privacy protocols is not safe while, with addition of some safeguards, differential privacy is preserved under finite precision up to some quantified inherent leakage.
Next, we propose a method to analyze programs that cannot be analyzed by a compositional analysis due to "erratic" control flow. This method based on rewrite system techniques allows us to use the proof of correction of the program in the exact semantics to prove the program is still safe in the finite representation.